Parallel Computing

Hwansoo Han (SKKU)
Unicore Limitations

- Performance scaling stopped due to
  - Power consumption
  - Wire delay
  - DRAM latency
  - Limitation in ILP
**CELL Processor – 1 + 8 cores**

- **One 64-bit PowerPC**
  - 3.2 GHz, Dual issue, two threads, 512KB L2 cache
  - 256 GFLOPS

- **Eight Synergistic Processors**
  - SIMD streaming processors, 256KB local store
Cell Applications

- Cell powered products
  - Game consoles
  - HPC servers
  - HDTVs

IBM Cell Blade Server

Sony PS3

Toshiba Cell Regza TV
GPGPU – Nvidia G80

- **G80 (GeForce 8800GTX)**
  - 128 stream processors, 518 GFLOPS (single precision FP)
  - Double precision FP will be available on Fermi (540 cores)

- **CUDA**
  - Extended ANSI C for program APIs
CUDA Applications

- Applications
  - Ray-tracing
  - H.264 codec
  - High Performance Computing
Memory Architectures

- **Centralized memory**
  - Shared memory with uniform memory access time (UMA)
    - SMP (symmetric multiprocessors)
    - Multicores

- **Distributed memory**
  - Shared memory with non-uniform memory access time (NUMA)
    - CC-NUMA (HP Superdome, SGI Origin)
    - Distributed shared machine (DSM)
  - Message passing
    - Separate address space per CPU
HW Cache Coherence

- **Bus-based “Snooping” protocol (multicores)**
  - Broadcast data accesses to all processors
  - Processors snoop to check if they have a copy
  - Works well with bus

- **Directory-based protocol (CC-NUMA)**
  - Usually implemented for distributed memory
  - Keep track of sharing of cache lines in a directory
  - Use point-to-point request to get directory information and shared data
  - Scale better than snooping
Performance by Amdahl’s Law

- **Speedup estimation**
  - Improved $n$ times for the parallel portion ($P$)

  \[
  \text{Speedup} = \frac{1}{(1 - P) + P/n}
  \]

- **Speedup limit**
  - If $n \to \infty$ (improved to eliminate the portion),

  \[
  \text{Speedup} = \frac{1}{1 - P}
  \]
80% of sequential execution time is parallelizable \((P = 0.8)\)

- Ideally parallelized on quad-core \((n = 4)\)

\[
\text{Speedup} = \frac{1}{(1 - P) + P/n} = \frac{1}{(1 - 0.8) + 0.8/4} = 2.5
\]

- Speedup limit \((n \rightarrow \infty)\)

\[
\text{Speedup} = \frac{1}{1 - P} = \frac{1}{1 - 0.8} = 5
\]
After some improvement, dominant part is not dominant any more.
Gustafson’s Law

- **Reevaluating Amdahl’s Law**
  - which assumes fixed amount of work

- **New assumptions for scaling-up the problem**
  - Parallel computing is needed for larger data
  - Sequential portion is “relatively” fixed, as the problem size grows

\[
\text{Speedup} = \frac{s + np}{s + p} = s + n(1 - s)
\]

where \( s + p = 1 \)

- Speedup approaches \( n \), as \( s \to 0 \)
  - Sufficiently large problem can be efficiently parallelized
**Superlinear Speedup**

- **Original sequential performance is not optimized**
  - Thrashing – cache, memory, disk

- **More cores mean more caches**
  - Avoid thrashing by dividing the work to fit in the local cache
  - Often results in superlinear speedup

[Diagram showing CPU, L1, L2 caches and working set]
Load Imbalance

- Exactly even workload?
  - Dynamic events (cache miss, branch miss prediction, ...)
  - Algorithm itself

- Master/worker

- OpenMP schedule – dynamic, guided

![Diagram of load imbalance and improved scheduling](image.png)
**Parallelization Loops**

- **Data parallel loops**
  - Perform the same operations
  - On different data
  - SPMD (same program, multiple data)

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

```
N = 16
fork (threads)
  i = 0
  i = 1
  i = 2
  i = 3
  i = 4
  i = 5
  i = 6
  i = 7
  i = 8
  i = 9
  i = 10
  i = 11
  i = 12
  i = 13
  i = 14
  i = 15
join (barrier)
```
Parallelization Loops (cont’d)

- **Thread pool**
  - Use pool of threads without create/join
  - Reduce thread creation/join overhead

![Diagram of thread pool and idle threads]
Reduction

- **Reduction analysis**
  - Associative operations
  - Intermediate result is never used in other places within loop

- **Parallel reduction**
  - Accumulate Partial results
  - Combine the partial results into one

```plaintext
for i = 1 to N
    sum = sum + A[i]
```
**SW Pipelining**

- Can handle loops with loop-carried dependences
  - Initiation interval
  - Regular patterns across multiple iterations
Programming Model

- **Shared memory**
  - Thread Library
    - POSIX thread, Java thread, ...
  - OpenMP
    - directives for compilers

- **Distributed memory**
  - MPI (message passing interface)
    - Communication API among multiple processes
  - Proprietary APIs
    - Cell B/E APIs, ...

- Small number of cores: < 10
- Not scalable for many cores
Implementation on Shared Memory

- **Thread Library**
  - Library calls
  - Low level programming
    - Explicit thread creation & work assignment
    - Explicit handling of synchronization
  - Parallelism expression
    - Task: create/join thread
    - Data: detailed programming
  - Design concurrent version from the start

- **OpenMP**
  - Compiler directives
  - Higher abstraction
    - Compilers convert code to use OpenMP library, which is actually implemented with thread APIs
  - Parallelism expression
    - Task: task/taskwait, parallel sections
    - Data: parallel for
  - Incremental development
    - Start with sequential version
    - Insert necessary directives
Implementation Examples

- **Threaded functions**
  - Exploit data parallelism

```c
node A[N], B[N];

main() {
    for (i=0; i<nproc; i++)
        thread_create(par_distance);
    for (i=0; i<nproc; i++)
        thread_join();
}
void par_distance() {
    tid = thread_id();   n = ceiling(N/nproc);
    s = tid * n;         e = MIN((tid+1)*n, N);
    for (i=s; i<e; i++)
        for (j=0; j<N; j++)
            C[i][j] = distance(A[i], B[j]);
}
```

- **Parallel loops**
  - Exploit data parallelism

```c
node A[N], B[N];

#pragma omp parallel for
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        C[i][j] = distance(A[i], B[j]);
```
MPI (message passing interface)
- Language independent communication library
- Freely available implementation
  - MPICH (Argonne Lab), Open MPI

```c
/* processor 0 */
node A[N], B[N];

Dist_calc0() {
    Send (A[N/2 .. N-1], B[0 .. N-1]);

    for (i=0; i<N/2; i++)
        for (j=0; j<N; j++)
            C[i][j] = distance(A[i], B[j]);

   Recv (C[N/2 .. N-1][0 .. N-1]);
}

/* processor 1 */
node A[N], B[N]; /* duplicate copies */

Dist_calc1() {
    Recv (A[N/2 .. N-1], B[0 .. N-1]);

    for (i=N/2; i<N; i++)
        for (j=0; j<N; j++)
            C[i][j] = distance(A[i], B[j]);

    Send (C[N/2 .. N-1][0 .. N-1]);
}
```
Summary

- **Parallel Architectures**
  - Multicores
  - CC-NUMA
  - Explicitly managed memories (Cell, GPGPU)

- **Parallelism in Applications**
  - Amdahl’s law, Gustafson’s law
  - Superlinear speedup, load balance
  - Parallelism in loops
    - Parallel loop, reduction, SW pipelining
  - Programming models
    - Pthread, OpenMP for shared memories
    - MPI, proprietary APIs for distributed memories