HPC – Algorithms and Applications

Fundamentals – Parallel Architectures, Models, and Languages

Michael Bader

Winter 2013/2014
Part I

Parallel Architectures
Multicore CPUs – Intel’s Nehalem Architecture

- example: quad-core CPU with shared and private caches
- simultaneous multithreading: 8 threads on 4 cores
- memory architecture: Quick Path Interconnect (replaced Front Side Bus)
Multicore CPUs – Intel’s Nehalem Architecture (2)

- NUMA (non-uniform memory access) architecture: CPUs have “private” memory, but uniform access to remote memory
- max. 25 GB/s bandwidth

**Intel® QuickPath Technology**

(source: Intel – Nehalem Whitepaper)
Manycore CPU – Intel Knights Ferry

Intel® MIC Architecture:
An Intel Co-Processor Architecture

Many cores and many, many more threads
Standard IA programming and memory model

(source: Intel/K. Skaugen – SC’10 keynote presentation)
Manycore CPU – Intel Knights Ferry (2)

Knights Ferry

- Software development platform
- Growing availability through 2010
- 32 cores, 1.2 GHz
- 128 threads at 4 threads / core
- 8MB shared coherent cache
- 1-2GB GDDR5
- Bundled with Intel HPC tools

Software development platform for Intel® MIC architecture

(source: Intel/K. Skaugen – SC’10 keynote presentation)
CUDA’s hierarchy of threads maps to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores and other execution units in the SM execute threads. The SM executes threads in groups of 32 threads called a warp. While programmers can generally ignore warp execution for functional correctness and think of programming one thread, they can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses.

The first Fermi based GPU, implemented with 3.0 billion transistors, features up to 512 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The 512 CUDA cores are organized in 16 SMs of 32 cores each. The GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distributes thread blocks to SM thread schedulers.

Fermi’s 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

(source: NVIDIA – Fermi Whitepaper)
The third generation SM introduces several architectural innovations that make it not only the most powerful SM yet built, but also the most programmable and efficient.

CUDA Core
- Dispatch Port
- Operand Collector
- FP Unit
- INT Unit
- Result Queue

Fermi Streaming Multiprocessor (SM)
- Warp Scheduler
- Dispatch Unit
- Register File (32,768 x 32-bit)
- Interconnect Network
- 64 KB Shared Memory / L1 Cache
- Uniform Cache

(source: NVIDIA – Fermi Whitepaper)
GPGPU – NVIDIA Fermi (3)

General Purpose Graphics Processing Unit:

- 512 CUDA cores
- improved double precision performance
- shared vs. global memory
- new: L1 und L2 cache (768 KB)
- trend from GPU towards CPU?

Fermi Memory Hierarchy

[Diagram showing memory hierarchy with shared memory, L1 cache, L2 cache, and DRAM]
Future Parallel Computing Architectures?

Not exactly sure how the hardware will look like . . . (CPU-style, GPU-style, something new?)

**However:** massively parallel programming required

- revival of vector computing
  → several/many FPUs performing the same operation
- hybrid/heterogenous architectures
  → different kind of cores; dedicated accelerator hardware
- different access to memory
  → cache and cache coherency
  → small amount of memory per core
- new restrictions → power efficiency, heat management, . . .
Part II

Parallel Models
The PRAM Model(s)

Concurrent or Exclusive Read/Write Access:
- **EREW**: exclusive read, exclusive write
- **CREW**: concurrent read, exclusive write
- **ERCW**: exclusive read, concurrent write
- **CRCW**: concurrent read, concurrent write
Exclusive/Concurrent Read and Write Access

- **Exclusive read**
  - Diagram:
    - X1, X2, X3, X4, X5, X6

- **Concurrent read**
  - Diagram:
    - X, Y

- **Exclusive write**
  - Diagram:
    - X1, X2, X3, X4, X5, X6

- **Concurrent write**
  - Diagram:
    - X, Y

Michael Bader: HPC – Algorithms and Applications
Fundamentals – Parallel Architectures, Models, and Languages, Winter 2013/2014
Example: Minimum Search on the PRAM

“Binary Fan-In”:

```
4 7 3 9 5 6 10 8
4 3 5 8
3 5
3
```
Minimum on the PRAM – Implementation

MinimumPRAM( L:Array[1..n]) : Integer {
    ! n assumed to be 2^k
    ! Model: EREW PRAM
    for i from 1 to k do {
        for j from 1 to n/(2^i) do in parallel
            if L[2j−1] > L[2j]
                then L[j] := L[2j];
            else L[j] := L[2j−1];
        end if;
    }
    return L[1];
}

Complexity: \( T(n) = \Theta(\log n) \) on \( \frac{n}{2} \) processors
Parallel External Memory – Memory Scheme

[Arge, Goodrich, Nelson, Sitchinava, 2008]
Parallel External Memory – History

Extension of the classical I/O model:

- large, global memory (main memory, hard disk, etc.)
- CPU can only access smaller working memory (cache, main memory, etc.) of $M$ words each
- both organised as cache lines of size $L$ words
- algorithmic complexity determined by memory transfers

Extension of the PRAM:

- multiple CPUs access global shared memory (but locally distributed)
- EREW, CREW, CRCW classification (for local and external memory)
- similar programming model (synchronous execution, e.g.)
Compute-Bound vs. Memory-Bound Performance

Consider a memory-bandwidth intensive algorithm:

- you can do a lot more flops than can be read from memory
- computational intensity of a code: number of performed flops per accessed byte

Memory-Bound Performance:

- computational intensity smaller than critical ratio
- you could execute additional flops “for free”
- speedup only possible by reducing memory accesses

Compute-Bound Performance:

- enough computational work to “hide” memory latency
- speedup only possible by reducing operations
The Roofline Model

- peak stream bandwidth
- SpMV
- 5-pt stencil
- matrix mult. (100x100)
- peak FP performance
- without vectorization
- without instruction-level parallelism
- without NUMA
- non-unit stride

GFlop/s − log
Operational Intensity [Flops/Byte] − log

1/8 1/4 1/2 1 2 4 8 16 32 64

[Williams, Waterman, Patterson, 2008]
Bulk Synchronous Parallelism

- suggested as a “bridging model” between software and hardware aspects
- hardware model:
  - multiple CPUs with private memory
  - CPUs connected via point-to-point network
Bulk Synchronous Parallelism – Execution Model

Computation organised into sequence of “Super Steps”:

1. each CPU executes a sequence of operations (on local data, synchronised from the last super step)

2. CPUs send and receive point-to-point messages (no broadcasts or send/receive to/by multiple CPUs allowed)

3. synchronisation at a barrier

Goal:

- estimate time for steps 1, 2, 3 based on CPU speed, bandwidth, and latency
**Interconnection Networks**

- multiple CPUs with private memory
- CPUs connected via interconnection network
- new: topology of the network explicitly considered

Example: 1D mesh (linear array) of processors:
2D Processor Mesh (Array)

Problem: Broadcast

- information transported by at most 1 processor per step
- phase 1: propagate along first line
- phase 2: propagate along columns
Broadcast 2Dmesh \((P[1,1]:X, n)\) {

\[\begin{array}{l}
\text{! Model: 2D mesh } p[i,j] \text{ with } n \times n \text{ processors} \\
\text{input: } P[1,1]:X \text{ ! element to be broadcasted} \\
\text{for } j \text{ from } 1 \text{ to } n-1 \text{ do} \\
\quad P[1,j+1]:X <<< P[1,j]:X \\
\text{for } i \text{ from } 1 \text{ to } n-1 \text{ do} \\
\quad \text{for } P[i,j]: 1 \leq j \leq n \text{ do in parallel} \\
\quad \quad P[i+1,j]:X <<< P[i,j]:X \\
\text{end in parallel} \\
\text{output: } P[i,j]:X \text{ ! } X \text{ available on each processor} \\
\end{array}\]

\textbf{Time complexity:} \(2n - 2\) steps on an \(n \times n\) mesh of processors
Performance Evaluation – Speed-Up

Definition:

- \( T(p) \): runtime on \( p \) processors
- speed-up \( S(p) \) quantifies the improvement factor in processing speed:

\[
S(p) := \frac{T(1)}{T(p)}, \quad \text{typically: } 1 \leq S(p) \leq p
\]

Absolute vs. Relative Speed-Up:

- absolute speed-up: best sequential algorithm for the mono-processor system is compared to the best parallel algorithm for the multi-processor system
- relative speed-up: compare the same (parallel) algorithm on mono- and multi-processor system
Parallel Efficiency

Definition:

-效率 $E(p)$ 关系到加速比 $S(p)$ 和处理器数量 $p$:

$$E(p) := \frac{S(p)}{p}$$

-表明了处理速度的相对改进
-通常：$0 \leq E(p) \leq 1$
-再次：绝对效率 vs. 相对效率
Scalability

- reduction of execution time, if the number of processors is increased
- quantitative: speed-up or parallel efficiency
- *strong scalability*: increase number of processors for fixed problem size
- *weak scalability*: increase number of processors and increase problem size
- qualitative: is there an improvement at all?

“If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?”

(Seymour Cray)
Amdahl’s Law

Assumptions:

- program consists of a sequential part \( s \), \( 0 \leq s \leq 1 \), which can not be parallelised (synchronisation, data I/O, etc.)
- parallelisable part, \( 1 - s \), can be perfectly parallelised (perfect speed-up on arbitrary number of processors)
- execution time for the parallel program on \( p \) processors:

\[
T(p) = s \cdot T(1) + \frac{1 - s}{p} \cdot T(1)
\]
Amdahl’s Law (2)

• resulting speed-up:

\[
S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{s \cdot T(1) + \frac{1-s}{p} \cdot T(1)} = \frac{1}{s + \frac{1-s}{p}}
\]

• consider increasing number of processors:

\[
\lim_{p \to \infty} S(p) = \lim_{p \to \infty} \frac{1}{s + \frac{1-s}{p}} = \frac{1}{s}
\]

• Amdahl’s law: speed-up is bounded by \( S(p) \leq \frac{1}{s} \)

• message: any inherently sequential part will destroy scalability once the number of processors becomes big enough
Gustafson’s Law

Assumptions:

- Amdahl: sequential part stays for increased problem size
- Gustavson: assume that any sufficient large problem can be efficiently parallelised
- fixed-time concept:
  - parallel execution time is normalised to $T(p) = 1$
  - this contains a non-parallelisable part $\sigma$, $0 \leq \sigma \leq 1$
- execution time on the mono-processor:
  
  $$T(1) = \sigma + p \cdot (1 - \sigma)$$

- thus: sequential part of total work gets smaller with increasing $p$
Gustafson’s Law (2)

- resulting speed-up (as $T(p) = 1$):

$$S(p) = \sigma + p \cdot (1 - \sigma) = p - \sigma(p - 1)$$

- resulting parallel efficiency:

$$E(p) = \frac{S(p)}{p} = \frac{\sigma}{p} + (1 - \sigma) \rightarrow 1 - \sigma$$

- more realistic: larger problem sizes, if more processors are available; parallelisable parts typically increase
Part III

Parallel Languages
OpenMP

- shared-memory application programming interface (API)
- extends *sequential* programs by directives to help compiler generate parallel code
- available for Fortran or C/C++
- *fork-join-model*: programs will be executed by a team of cooperating threads
- memory is shared between threads, except few private variables
Matrix-Vector Product in OpenMP

```c
void mvp(int m, int n, double* restrict y,
         double** restrict A, double* restrict x)
{
    int i, j;
    #pragma omp parallel for default(none) \ 
        shared(m,n,y,A,x) private(i, j)
    for (i=0; i<n; i++) {
        y[i] = 0.0;
        for (j=0; j<n; j++) {
            y[i] += A[i][j]*x[j];
        }
    } /*--- end of omp parallel for ---*/
}
```
OpenMP directives are inserted as \texttt{#pragma}:

\texttt{#pragma omp parallel for default(none) \shared(m,n,y,A,x) private(i, j)}

Advantages:

- directives will be ignored by compilers that do not support OpenMP
- sequential and parallel program in the same code!
- incremental programming approach possible (add parallel code sections as required)
OpenMP’s Memory Model

```
#pragma omp parallel ... default(none) \ 
    shared(m,n,y,A,x) private(i, j)
```

- memory usually shared between threads – here: matrix and vectors
- however, certain variables are private: loop variables (here: indices), temporary variables, etc.
- if not specified, default settings apply – here: `default(none)` to switch off all default settings
- programmer is responsible to sort out concurrent accesses! (even if default settings are used)
Pthreads

- standardised programming interface according to POSIX (Portable Operating System Interface)
- thread model is a generalised version of the UNIX process model (forking of threads on shared address space)
- scheduling of threads to CPU cores done by operating system
Pthreads – Typical Methods

- `pthread_create(...)`: (main) thread creates a further thread that will start by executing a specified function
- `pthread_join(...)`: thread will wait until a specified thread has terminated (useful for synchronisation)
- `pthread_cancel(...)`: cancel another thread
- Functions to synchronise data structures:
  - `mutex`: mutual exclusive access to data structures;
  - `cond`: wait for or signal certain conditions
- Functions to take influence on scheduling
- Etc.
Programming Patterns

Pthreads allow arbitrary coordination of threads; however, certain programming patterns are common, e.g.:

- **Master-Slave** model:
  master thread controls program execution and parallelisation by delegating work to slave threads

- **Worker** model:
  threads are not hierarchically organised, but distribute/organise the operations between themselves (example: jobs retrieved from a work pool)

- **Pipelining** model:
  threads are organised via input/output relations: certain threads provide data for others, etc.
Example: Matrix Multiplication

```c
#include <pthread.h>
typedef struct {
    int size, row, column;
    double (*MA)[8], (*MB)[8], (*MC)[8];
} matrix_type_t;

void thread_mult(matrix_type_t *work) {
    int i, row = work->row, col = work->column;
    work->MC[row][col] = 0.0;
    for (i=0; i < work->size; i++)
        work->MC[row][col] +=
            work->MA[row][i] * work->MB[i][col];
}
```
Example: Matrix Multiplication (cont.)

```c
void main() {
    double MA[8][8], MB[8][8], MC[8][8];
    pthread_t thread[8*8];
    for(int row=0; row<8; row++)
        for(int col=0; col<8; col++) {
            matrix_type_t *work = (matrix_type_t *) malloc( /* ... */ );
            work->size = 8; work->row = row; work->col = col;
            work->MA = MA; work->MB = MB; work->MC = MC;
            pthread_create(&thread[col+8*row], NULL, (void*) thread_mult, (void*) work);
        }
    for(int i=0; i<8*8; i++) pthread_join(thread[i], NULL);
}
```

(example from: Rauber&Rünger: Parallele Programmierung)
Java Threads

- object-oriented language design explicitly includes threads
- class `Thread` to represent threads – can be extended to implement customised threads (inherits `start()`, `run()` methods, etc.)
- interface `Runnable`:
  - classes that implement `Runnable`, i.e., provide a method `run()` can be used to create a thread:
    ```java
    Thread th = new Thread(runnableObject);
    ```
- keyword `synchronized` for methods that should be treated as a critical region
- methods `wait()` and `notify()` in class `Object`
Example: Matrix Multiplication

class MatMult extends Thread {
    static int a [], b [], c [], n=3;
    int row;
    MatMult(int _row) {
        row = _row; this.start();
    }
    public void run() {
        for(int i=0; i<n; i++) {
            c[row][i] = 0.0;
            for(int j=0; j<n; j++)
                c[row][i] = c[row][i] + a[row][j]*b[j][i];
        }
    } /* class MatMult t.b.c. */
public static void main() {
    a = new int[n][n]; b = new int[n][n]; c = new int[n][n];
    /* ... initialise a, b, c ... */
    MatMult mat = new MatMult[n];
    for (int i=0; i<n; i++) mat[i] = new MatMult(i);
    try {
        for (int i=0; i<n; i++) mat[i].join();
    } catch (Exception E) { /* ... */ }
}

(cmp. example in Rauber&Rünger: Parallele Programmierung)
Unified Parallel C (UPC)

- extension of C, specified in the ISO C99 standard
- based on a distributed shared memory model: physically distributed memory with shared address space
- **PGAS** language: “partitioned global address space”
- single program, multiple data: every program is executed in parallel on specified number of threads
- variables are private by default, but can be declared as shared
- consistency model can be varied: strict vs. relaxed
Example: Matrix Multiplication

Declaration of variables:

\[
\text{shared } [N\times N/\text{THREADS}] \text{ int } a[N][N]; \\
\text{shared } [N/\text{THREADS}] \text{ int } b[N][N]; \\
\text{shared } [N\times N/\text{THREADS}] \text{ int } c[N][N];
\]

Variables have an affinity towards threads:

- \textit{block-cyclic} distribution of variables top threads
- \texttt{a} and \texttt{c} declared with a block size of \(N\times N/\text{THREADS}\) → block-oriented distribution of rows to threads
- \texttt{b} declared with a block-size of \(N/\text{THREADS}\) → block-oriented distribution of columns to threads

affinity can reflect physical distribution of data
Example: Matrix Multiplication (cont.)

Code excerpt for matrix multiplication:

```upc
forall ( i=0; i<N; i++ ; &a[i][0])
    /* &a[i][0] specifies that iteration will be executed by thread that has affinity to a[i][0] */
    for (j=0; j<N; j++) {
        c[i][j] = 0;
        for (l=0; l<N; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
upc_barrier;
```

(source: Rauber&Rünger: Parallele Programmierung)
Further PGAS Languages

- Co-Array Fortran (CAF)
  → will become part of the next Fortran standard
- Titanium (similar to UPC, but for Java)
- X10: extension of Java;
  *globally asynchronous, locally synchronous*: add “places” that execute threads
- Chapel
- Fortress
Further Example: Intel PBB

“Intel Parallel Building Blocks”:

- language extensions & libraries for C/C++
- **Intel Cilk Plus**: language extension for simple loop & task oriented parallelism for C/C++
- **Intel Threading Building Blocks**: C++ template library to support task parallelism
- **Intel Array Building Blocks**: C++ template library to support vector parallelism
Examples – Intel Cilk Plus

**Intel Cilk:**

```c
void mergesort(int a[], int left, int right) {
    if (left < right) {
        int mid = (left + right)/2;
        cilk_spawn mergesort(a, left, mid);
        mergesort(a, mid, right);
        cilk_sync;
        merge(a, left, mid, right);
    }
}
```

(source: Intel)
Examples – Intel ArBB

**Intel ArBB:**

define void matvec_product(const dense<f32, 2>& matrix, const dense<f32>& vector, dense<f32>& result)
{
    result = add_reduce(matrix
        * repeat_row(vector, matrix.num_rows()));
}

(source: Intel)
Abstraction of a distributed memory computer

- MPI run consists of a set of processes with separate memory space
- processes can exchange data by sending messages
- no explicit view on network topology required

SIMD/MIMD?? → MPMD/SPMD

- processes can run different programs ("codes")
  → multiple program multiple data (MPMD)
- more common (and simpler):
  processes run instances of the same program ("code")
  → single program multiple data (SPMD)
MPI Example: “Hi there” . . .

```c
#include "mpi.h"
int main( int argc, char **argv )
{
    int myrank;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0)
        send_a_message();
    else if (myrank == 1)
        receive_a_message();
    MPI_Finalize();
}
```
MPI Example: “Hi there” . . . (cont.)

```c
void send_a_message() {
    char message[40];
    strcpy(message,"Mr. Watson, come here, I want you.");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1,
             110, MPI_COMM_WORLD);
}

void receive_a_message() {
    char message[40];
    MPI_Status status;
    MPI_Recv(message, 40, MPI_CHAR, 0,
             110, MPI_COMM_WORLD, &status);
    printf ("received: %s:\n", message);
}
```
References (Languages)