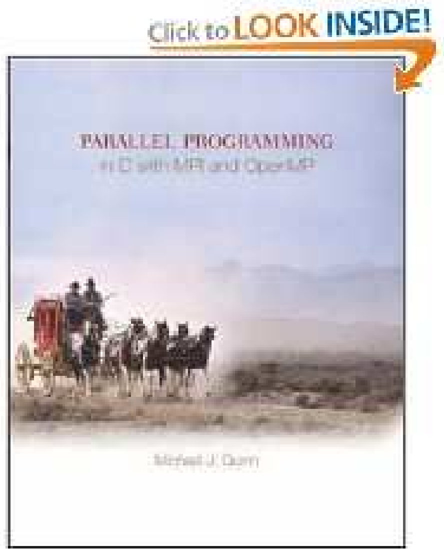
Summary of INF3380

Content

- General topics on parallelization and parallel programming
- MPI programming
- OpenMP programming
- Performance analysis
- Applications

About the exam

- 4-hour written exam
- One A4-sheet with notes and a calculator are allowed to take to the exam
- Syllabus: Chapters 3,4,5,6,7,11,13,14,17,18 from the textbook, plus all the lecture slides



- Chapters 1 & 2 provide important background info

Parallel computing and programming in general

- Parallel computing – a form of parallel processing by utilizing multiple computing units concurrently for one computational problem
  - shortening computing time
  - solving larger problems
- Manual parallel programming is needed because
  - Modern multicore-based computers are good at multi-tasking, but not good at automatically computing one problem in parallel
  - Automatic parallelization compilers have had little success
  - Special parallel programming languages have had little success
- Parallel programming = serial programming + finding parallelism + enforcing work division and collaboration

Foster's design methodology

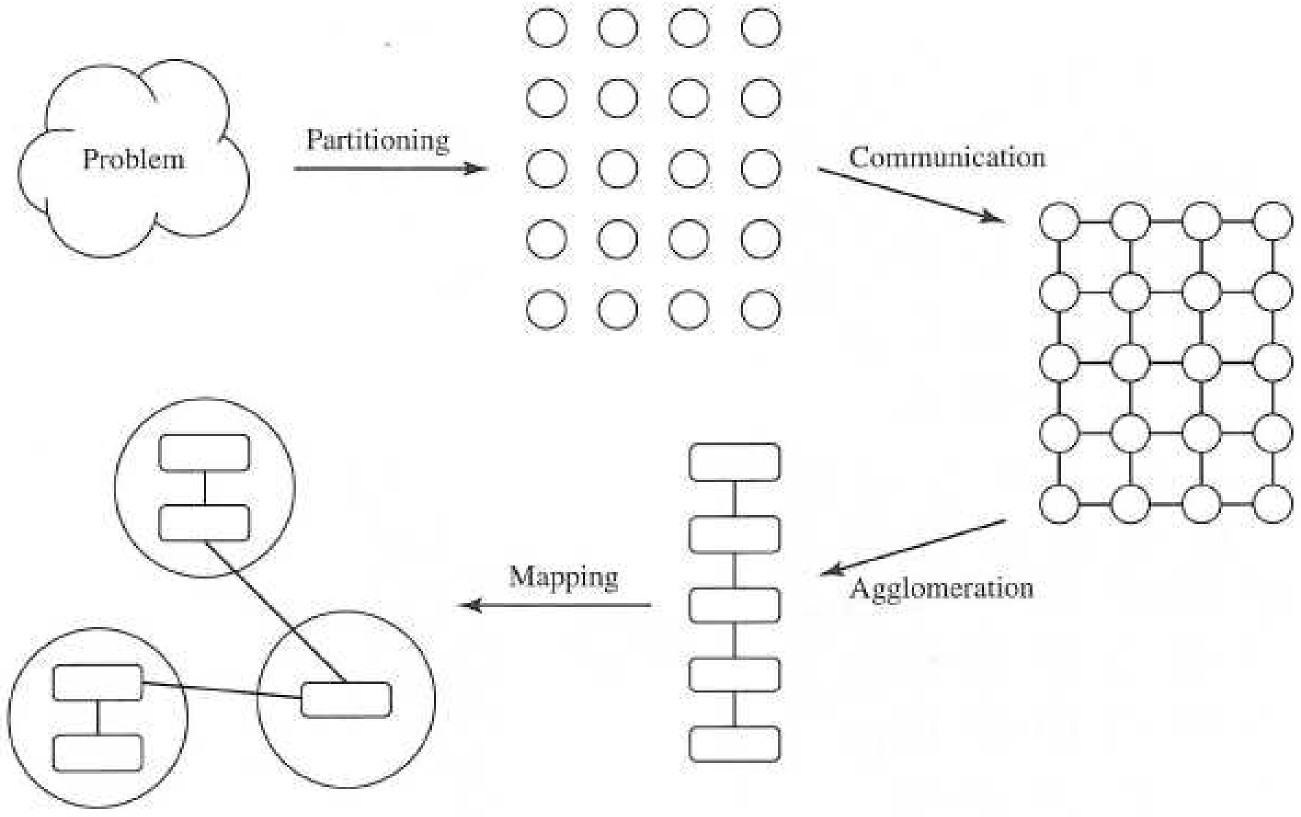


Figure 3.2 Foster's parallel algorithm design methodology.

A four-step process for designing parallel algorithms

I. Foster, **Designing and Building Parallel Programs**, Addison-Wesley, 1995

Speed of parallel computations

- Parallel computations = each processor simultaneously does its assigned computations + collaboration between the processors
- Speed of the computations on each processor mostly depends on effective use of memory and cache — data locality is very important
- Speed of a parallel program depends on
  - as much work as possible can be divided among processors
  - the work load division is even among the processors
  - each processor finishes its computations quickly
  - low overhead of parallelization-specific computations
  - low overhead of “collaboration cost” between the processors
    - synchronization
    - communication

Message passing programming

- Assumption: each processor's own memory is not directly accessible by other processors
- Collaboration between the processors is through sending and receiving messages between the processors
  - a message is an array of predefined data types
  - point-to-point communication
  - collective communication
- The global data structure is normally divided among the processors (not duplicated)
- MPI is the de-facto standard of message passing programming

MPI basics

- The working units are called MPI processes
- An MPI communicator is group of processes
- Each process within a communicator has a unique rank, between `0` and `#procs-1`
- Carelessly programmed MPI communications may deadlock
- Non-deterministic features of an MPI program
  - Between communications, the different processes may proceed at different paces
  - If a process is expecting two messages from two senders, the order of arrival is normally not known beforehand
- Synchronization
  - explicit – `MPI_Barrier`
  - implicit – collective commands or matching `MPI_Send` and `MPI_Recv`

Overlap communication with computation

- Performance may be improved on many systems by overlapping communication with computation
- Use of non-blocking and completion routines
- For example, initiate the communication with `MPI_Isend` and `MPI_Irecv`, continue with computation, finish with `MPI_Wait`

Thread programming for shared memory

- Thread programming is a natural model for shared memory
  - Execution unit: thread
  - Many threads have access to shared variables
  - Information exchange is (implicitly) through the shared variables
- OpenMP is the most user-friendly thread programming standard

The programming model of OpenMP

- Multiple cooperating threads are allowed to run simultaneously
- The threads are created and destroyed dynamically in a **fork-join** pattern
  - An OpenMP program consists of a number of parallel regions
  - Between two parallel regions there is only one master thread
  - In the beginning of a parallel region, a team of new threads is spawned
  - The new threads work simultaneously with the master thread
  - At the end of a parallel region, the new threads are destroyed

The memory model of OpenMP

- Most variables are shared between the threads
- Each thread has the possibility of having some private variables
    - Avoid race conditions
    - Passing values between the sequential part and the parallel region

OpenMP basics

- `#pragma omp parallel` starts a parallel region
  - all the threads execute the same code (if nothing else is said)
- `#pragma omp for` divides the work of `for`-loop between the threads
  - each thread does a subset of the iterations
  - the actual division of the iterations depends on the scheduler
- `#pragma omp sections` can used for task parallelism

Non-parallel execution among threads

- `#pragma omp single` { `...` }
- `#pragma omp master` { `...` }
- `#pragma omp critical` { `block of codes` }
- `#pragma omp atomic` { `only one statement` }
- `#pragma omp barrier`

Overhead in OpenMP programs

- Creation and termination of threads
- Scheduling of threads in connection with `#pragma omp for`
- (Invisible) synchronization
- (Invisible) copy cost between shared and private variables
- Serialized execution in parallel regions

Things to remember

- First step: identify parallelism in a sequential algorithm
  - find out the operations that can be done simultaneously
- Good work division is important
  - even distribution of the work load among processors
  - keep the overhead of resulting communication low
- On distributed memory, data should be divided as well
- Be aware of needed synchronizations (both MPI and OpenMP)
- Be aware of possible deadlocks (both MPI and OpenMP)
- Be aware of possible racing conditions (OpenMP)

Performance analysis

Basic questions:

- How to roughly predict computing time as function of the number of processors?
- How to analyze parallel execution times?
- When does it pay off to use more processors?

Important notation and definitions

n	problem size
p	number of processors
$\sigma(n)$	inherently sequential computation
$\varphi(n)$	parallelizable computation
$\kappa(n,p)$	parallelization overhead

$$\text{Speedup } \Psi(n,p) = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

$$\text{Efficiency } \varepsilon(n,p) = \frac{\text{Sequential execution time}}{\text{Processors used} \times \text{Parallel execution time}}$$

Observations

- Sequential execution time $= \sigma(n) + \varphi(n)$
- Parallel execution time $\geq \sigma(n) + \varphi(n)/p + \kappa(n,p)$

$$\text{Speedup } \Psi(n,p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n,p)}$$

$$\text{Efficiency } \varepsilon(n,p) \leq \frac{\sigma(n) + \varphi(n)}{p\sigma(n) + \varphi(n) + p\kappa(n,p)}$$

Amdahl's Law

Suppose we neglect the parallel overhead $\kappa(n,p)$, and if we know the inherently sequential portion of the computation,

$$f = \frac{\sigma(n)}{\sigma(n) + \varphi(n)}$$

then, the best achievable speedup can be estimated as

$$\Psi \leq \frac{1}{f + (1-f)/p}$$

Upper limit (when p goes to infinity): $\Psi \leq \frac{1}{f+(1-f)/p} < \frac{1}{f}$

Gustafson–Barsis's Law

We may not know the computing time needed by a single processor, because the problem size is too big for one processor

However, suppose we know the fraction (s) of time spent by a parallel program (using p processors) on performing inherently sequential operations

$$s = \frac{\sigma(n)}{\sigma(n) + \varphi(n)/p}$$

$$\begin{aligned}
\Psi(n,p) &\leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p} \\
&= \frac{(s + (1-s)p)(\sigma(n) + \varphi(n)/p)}{\sigma(n) + \varphi(n)/p} \\
&= p + (1-p)s
\end{aligned}$$

Karp–Flatt Metric

Both Amdahl's Law and Gustafson–Barsis's Law ignore the parallelization overhead $\kappa(n,p)$

If we consider the parallelization overhead as another kind of "inherently sequential work", then we can use Amdahl's law to experimentally determine a "combined" serial fraction e, which is defined as

$$e(n,p) = \frac{\sigma(n) + \kappa(n,p)}{\sigma(n) + \varphi(n)}$$

The experimentally determined serial fraction $e(n,p)$ can be computed based on knowing $\Psi(n,p)$

$$e = \frac{1/\Psi - 1/p}{1 - 1/p}$$

Isoefficiency relation

- Purpose: to study scalability—the ability to maintain parallel efficiency $\varepsilon(n,p)$ when p is increased
- Problem size n must also increase with p, but how fast?
- Suppose we know the explicit formulas for $T(n,1)$ and $T(n,p)$
- We denote $T_o(n,p)$ as

$$T_o(n,p) = pT(n,p) - T(n,1) = (p-1)\sigma(n) + p\kappa(n,p)$$

- If we want to maintain $\varepsilon(n,p)$ when both p and n increase, we must have so big n such that

$$T(n,1) \geq \frac{\varepsilon}{1-\varepsilon} T_o(n,p)$$

The sieve of Eratosthenes

- Finding prime numbers
- Pseudocode:

```
1. Create a list of natural numbers $2, 3, 4, \ldots, n$, none is marked.
2. Set $k$ to 2, the first unmarked number on the list
3. Repeat
      (a) Mark all multiples of $k$ between $k^2$ and $n$
      (b) Find the smallest number greater than $k$ that is
   unmarked. Set $k$ to this new value.
      Until $k^2 > n$
4. The unmarked numbers are primes.
```

- Source of parallelism: Step (a) can be done concurrently by many processes, each responsible for a “segment” of the list

Floyd's algorithm

- Starting point: n vertices and adjacency matrix $a[i,j]$
- Algorithm:

```
for $k \leftarrow$ 0 to $n-1$
    for $i \leftarrow$ 0 to $n-1$
        for $j \leftarrow$ 0 to $n-1$
            $a[i,j] \leftarrow \min(a[i,j],\ a[i,k]+a[k,j])$
        endfor
    endfor
endfor
```

- Parallelism lies within each k iteration

Matrix multiplication

```
for (i=0; i<l; i++)
  for (j=0; j<n; j++) {
    C[i][j] = 0.;
    for (k=0; k<m; k++)
      C[i][j] += A[i][k]*B[k][j];
  }
```

- Parallelism: each entry of matrix C can be computed independently
- On a distributed-memory system, matrix A and matrix B can be either partitioned rowwise block-striped or checkerboard block decomposed

Parallel finite differences

- Algorithm example:

$$u_i^{\ell+1} = u_i^\ell + \kappa \frac{\Delta t}{\Delta x^2}\left(u_{i-1}^\ell - 2u_i^\ell + u_{i+1}^\ell\right) + \Delta t f(x_i, t_\ell)$$

- Computations of $u_i^{\ell+1}$ and $u_j^{\ell+1}$ are independent of each other
- We can divide the work of computing $u^{\ell+1}$ among MPI processes or OpenMP threads
- Blockwise work division in MPI
- Need for MPI communication to obtain values for ghost points

Quicksort

- Sequential quicksort:
  - Select one of the numbers as pivot
  - Divide the list into two sublists: a “low list” containing numbers smaller than the pivot, and a “high list” containing numbers larger than the pivot
  - The low list and high list recursively repeat the procedure to sort themselves
  - The final sorted result is the concatenation of the sorted low list, the pivot, and the sorted high list

Parallel quicksort algorithms

- Observation: the low list and high list can sort themselves concurrently
- Starting point for parallel quicksort on distributed memory:
  - The unsorted list is evenly distributed among the processes
- Desired result of a parallel quicksort algorithm:
  - The list segment stored on each process is sorted
  - The last element on process i's list is smaller than the first element on process $i+1$'s list
- Three parallel algorithms (see Chapter 14)