Advanced Computer Architecture II: Multiprocessor Design

Parallel Program Structure and Single Chip Parallelism

Professor Russ Joseph
Department of Electrical and Computer Engineering
Northwestern University

January 8, 2007

Parallel Programming

Look at parallel speedup and limitations.

Examine parallel program design.

Talk about on-chip parallelism.

Why Parallelize A Program?

We're not doing this purely for fun, we need speedup.

Obviously we can solve the same problems quicker.

...or we can solve the same problem with greater accuracy.

...or we can solve larger more interesting problems in the same amount of time.

Parallel Speedup

The speedup of a parallel implementation over a serial on can be represented as:

$$S_p = \frac{T}{T_p} \tag{1}$$

- T - time for serial implementation
- T_p - time for parallel implementation

The efficiency of the implementation is given by:

$$E_p = \frac{S_p}{p} = \frac{T}{pT_p} \tag{2}$$

- If efficency is 100%, then the speedup is linear.
- In most practical cases, the efficency never reaches 100%.

Amdahl's Law

Amdahl's Law expresses the limits on performance improvement:

$$T_p = T(\alpha + \frac{1-\alpha}{p}) \qquad (3)$$

- α - the fraction of the implementation that cannot be parallelized (serial portion)
- Speedup of a parallel implementation is limited by the fraction of time that parallelism cannot be exploited.

Limits to Parallelism

The serial portion of implementation, α, can have numerous causes.

We will take a closer look at these three:

- communication
- synchronization
- load imbalance

Communication

Usually some degree of information share among cooperating processors.

Sometimes, communication is explicit in algorithm.

Often, it is an artifact of hardware implementation.

Important concepts:

latency - time for an operation

bandwidth - rate at which operations are performed

cost - impact on program execution time

Generic Data Transfer

Generic linear model for data transfer:

$$transfer\ time(n) = T_0 + \frac{n}{B} \qquad (4)$$

- T_0 - the start-up cost
- n - the amount of data (bytes)
- B - the transfer rate

Half-power point - $n_{1/2} = T_0 B$.

Network Data Transfer

Communication in a network of parallel processors can be modeled as:

$$communication\ time(n) = overhead + occupancy + network\ delay \quad (5)$$

- *overhead* - time the processor spends initiating the transfer (processor is busy)
- *occupancy* - time for data to pass through slowest component
- *network delay* - remaining time in communication path

Communication Cost

Total impact of communication on performance (from the point of view of an initiating processor) can be represented as:

$$communication\ cost = frequency \times (communication\ time - overlap) \quad (6)$$

- *frequency of communication* - communication operations per unit of work
- *communication time* - average as discussed previously
- *overlap* - amount of time the processor can do useful work while communication is taking place

Synchronization

Cooperating processors must coordinate their efforts to ensure valid data is used in computation (preserve dependence order).

But synchronization is overhead not present in serial implementation.

Synchronization is often coupled closely to communication.

Load Imbalance

Full efficiency is reached when all processors are completely utilized (doing useful work) all the time.

To achieve this goal, we need to divide work equally among processors.

Sometimes difficult to do at compile time, may require dynamic balancing.

Reducing Performance Limiters

If the actual speedup doesn't match goals, who is to blame?

- Program: Does the program expose enough parallelism?
- System: Does the system limit the overhead associated with communication/synchronization?
- Program/System: Does the program implementation suit the system?

Start by looking at parallelization process...

Definitions

Some preliminary definitions:

- *task* - arbitrarily defined piece of work
- *process* - abstract computational thread which performs one or more tasks
- *processor* - physical hardware on which a process executes

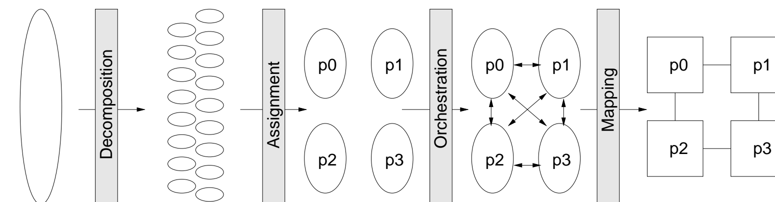
Tasks are performed by processes which execute on processors.

Parallelization

There are four components to the parallelization process:

- *decomposition* - dividing the serial computation into tasks
- *assignment* - appointing tasks to processes
- *orchestration* - arranging and organizing data access, communication, and synchronization among processes
- *mapping* - binding the processes to processors

Parallelization Diagram



Decomposition

Objective: Breaking the computation into a collection of tasks

Maximum number of tasks available at a given time sets upper bound on number of processes (and hence processors).

Limited concurrency is a fundamental limitation on achievable speedup.

Expose enough parallelism to maximize concurrency.

But if there are too many tasks, management could be significant compared to useful work.

Example

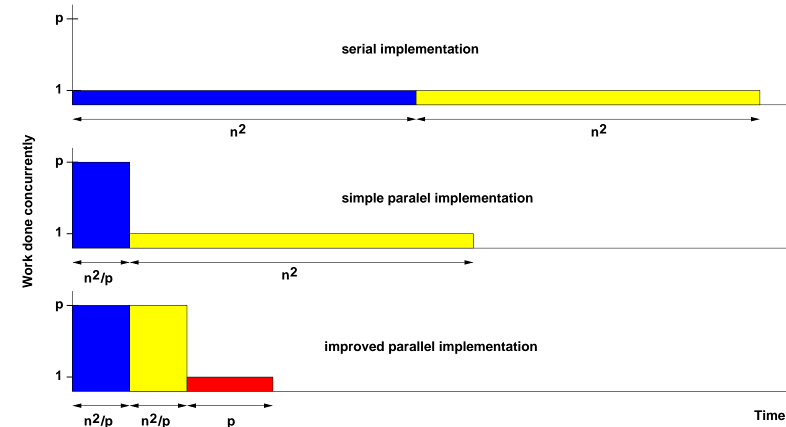
Consider a program which has two phases and is executing on p processors:

- Phase One: Perform a single operation on each of the elements of an n-by-n grid.
- Phase Two: Sum the individual points into a global sum value.

What is the execution time in terms of n and p?

Could this be improved?

Concurrency Profile



Assignment

Objective: Specify mechanism by which tasks are distributed.

Can be done on-line or off-line.

Try to balance workload among tasks.

Reduce interprocess communication.

Reduce run-time overhead in management.

Orchestration

Objective: Arrange and organize data access, communication, and synchronization among processes.

Programming model has large influence.

Try to organizing data structures efficiently.

Schedule tasks to exploit data locality.

Trade-off between communication granularity and frequency.

Specify communication and synchronization primitives.

Mapping

Objective: Bind processes to processors.

Usually very specific to system or programming environment.

Processors are partitioned into fixed subsets (e.g. space-sharing)

Program can bind or pin processes to processors.

Operating system may manage mapping entirely.

Performance Goals

Step	Architecture Dependent?	Performance Goals
Decomposition	Mostly no	Expose enough concurrency, but not too much
Assignment	Mostly no	Balance workload
		Reduce communication volume
Orchestration	Yes	Reduce non-inherent communication via data locality
		Reduce communication and synchronization cost (as seen by processor)
		Reduce serialization (at shared resources)
		Schedule tasks to satisfy dependencies early
Mapping	Yes	Put related processes on same processor if necessary
		Exploit locality in network topology

Equation Solver

A *kernel*, important subsegment of computation, for Ocean, a water current simulator.

- Based on 2D (n+2) by (n+2) grid array.
- Sweep across grid.
- Updates in-place (Gauss-Seidel method).
- Tests for convergence.

$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$

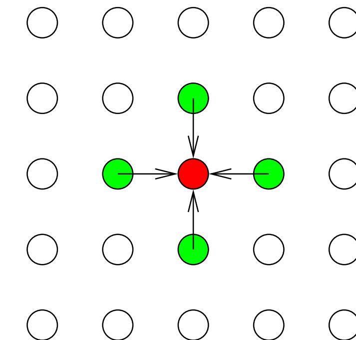


Figure 1: 2D Grid for Equation Solver.

Equation Solver: Loop Code

```
1  procedure Solve(a) /* solve the equation system */
2          float **A; /* A is (n+2) by (n+2) array */
3  begin
4          int i, j, done = 0;
5          float temp;
6          while (!done) do /* outermost loop over sweeps */
7                  diff = 0; /* initialize maximum difference to 0 */
8                  for i ← 1 to n do /* sweep over non-border points of grid */
9                          for j ← 1 to n do
10                                 temp = A[i,j]; /* save old value of element */
11                                 A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j]); /* compute average */
12                                 diff += abs(A[i,j] - temp);
13                         end for
14                 end for
15                 if (diff/(n*n) < TOL) then done = 1;
16         end while
17 end procedure
```


Equation Solver: Decomposition

Objective: Decompose work into individual grid points.

Change loop structure and loop over anti-diagonals.

Apply different computational order.

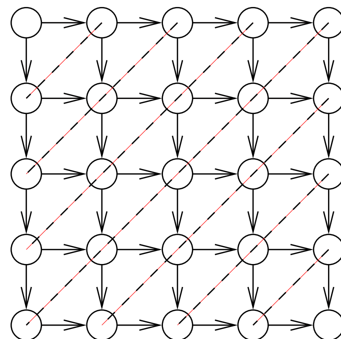


Figure 2: Dependence structure for Equation Solver.

Equation Solver: Red-Black Ordering

Red-Black Ordering performs computation in slightly different manner:

- Sweep is broken into two sub-sweeps.
- Exposes more parallelism.
- Different computation result with different convergence pattern.

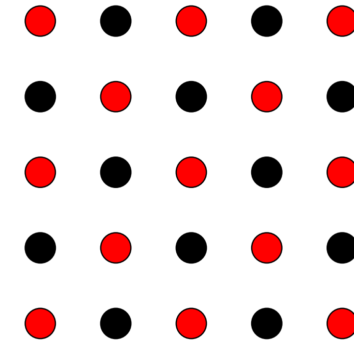


Figure 3: Red-Black ordering for Equation Solver.

Equation Solver: Simple Decomposition Strategy

In this example, we use a simple strategy:

- Ignore dependencies within a sweep.
- Can split computation into any set of aggregated groups of iterations.
- Here we will decompose into individual inner loop iterations.
- Reduced concurrency from n^2 to n.
- Approximately $2n$ words of communication for each set of n points.

Equation Solver: Assignment

Goal: Specify mechanism by which tasks are distributed.

Simple option is static with each process responsible for a set of rows (e.g. row i assigned to process $\lfloor \frac{i}{p} \rfloor$.

Another option is interleaved assignment (e.g. process i assigned rows $i, i+p, i+2p, \ldots$)

Could also dynamically assign rows to processes

Simple static choice achieves good load balance

Degree of parallelism reduced from n to p

Relatively small communication to computation ratio

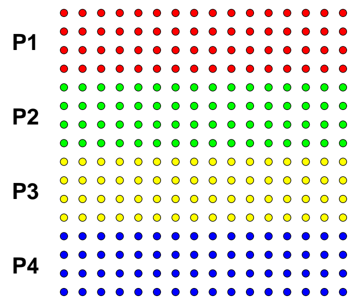


Figure 4: Simple static row set assignment.

Equation Solver: Orchestration

Goal: Arrange and organize data access, communication, and synchronization among processes.

Implementation highly dependent on programming model.

Parallel speedup dependent on programming model and architecture.

Review of Programming Models

Programming model is the conceptualization of the parallel machine that the programmer sees.

The three programming models we will use here are:

- data parallel
- shared address space
- message passing

Shared Address Space Model

Concept: Parallel threads share some portion of their virtual address space.

Analogy: Another company were everyone sits at a large table and freely looks at everyone else's on-going work.

Equation Solver: Initialization Code for Shared Address Space

```
1  int n, nprocs; /* size of matrix: (n+2 by n+2) elements */
2  float **A, diff = 0;
3  LOCKDEC (diff_lock);
4  BARDEC (bar1);
5  main()
6  begin
7          read(n); read(nprocs); /* read input parameter: matrix size */
8          A ← G_MALLOC (2D array);
9          initialize(A); /* initialize the matrix A */
10         CREATE(nprocs-1, Solve, A);
11         Solve(a); /* call routine to solve the equation */
12         WAIT_FOR_END(nprocs-1);
13 end main
```


Equation Solver: Loop Code for Shared Address Space

```
1  procedure Solve(a) /* solve the equation system */
2          float **A; /* A is (n+2) by (n+2) array */
3  begin
4          int i, j, pid, done = 0;
5          float temp, mydiff = 0;
6          int mymin = 1 + (pid * n/nprocs);
7          int mymax = mymin + n/nprocs - 1;
8          while (!done) do /* outermost loop over sweeps */
9                  mydiff = diff = 0; /* initialize maximum difference to 0 */
10                 BARRIER(bar1, nprocs);
11                 for i ← mymin to mymax do /* sweep over non-border points of grid */
12                         for j ← 1 to n do
13                                 temp = A[i,j]; /* save old value of element */
14                                 A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j]); /* compute average */
15                                 mydiff += abs(A[i,j] - temp);
16                         end for
17                 end for
18                 LOCK(diff_lock);
19                 diff += mydiff ;
20                 UNLOCK(diff_lock);
21                 BARRIER(bar1, nprocs);
22                 if (diff/(n*n) < TOL) then done = 1;
23         BARRIER(bar1, nprocs);
24         end while
25 end procedure
```


Equation Solver: Comments on Shared Address Space Orchestration

This example used both mutual exclusion and event synchronization.

Not too different from sequential code.

Equation Solver: Mapping

Goal: Bind processes to processors.

We will assume that the number of processes equals the number of processors.

No processors or processors are added or removed from the system.

The operating system places processes on processors in no particular order.

No migration.

Parallelization: Big Picture

As you can see, there are many steps between having a parallel algorithm and being able to run a parallel application.

There are numerous design choices which can have a large impact on performance.

Great! We know all there is to know about parallel programming.

What kinds of machines will get to play around with?

Summary

Parallel speedup can be limited by several factors.

Parallelization process has four steps:

- decomposition
- assignment
- orchestration
- mapping

Performance of an algorithm is highly dependent on implementation choices.