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 Programming

Look at parallel speedup and limitations.
Examine parallel program design.
Review programming models.

Parallel Speedup

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

\[ S_p = \frac{T}{T_p} \]  

- \( T \) - time for serial implementation
- \( T_p \) - time for parallel implementation

The efficiency of the implementation is given by:

\[ E_p = \frac{S_p}{p} = \frac{T}{pT_p} \]  

- If efficiency is 100%, then the speedup is linear.
- In most practical cases, the efficiency never reaches 100%.
Amdahl's Law

Amdahl's Law expresses the limits on performance improvement:

\[ T_p = T(\alpha + \frac{1 - \alpha}{p}) \]  \hspace{1cm} (3)

- \( \alpha \) - 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.

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:
- \textit{latency} - time for an operation
- \textit{bandwidth} - rate at which operations are performed
- \textit{cost} - impact on program execution time

Limits to Parallelism

The serial portion of implementation, \( \alpha \), can have numerous causes.
We will take a closer look at these three:
- communication
- synchronization
- load imbalance

Generic Data Transfer

Generic linear model for data transfer:

\[ \text{transfer time}(n) = T_0 + \frac{n}{B} \]  \hspace{1cm} (4)

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

\textit{Half-power point} - \( n_{1/2} = T_0B \).
**Network Data Transfer**

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

\[
\text{communication time}(n) = \text{overhead} + \text{occupancy} + \text{network delay} \tag{5}
\]

- **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:

\[
\text{communication cost} = \text{frequency} \times (\text{communication time} - \text{overlap}) \tag{6}
\]

- **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

<table>
<thead>
<tr>
<th>Step</th>
<th>Architecture Dependent?</th>
<th>Performance Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition Assignment</td>
<td>Mostly no</td>
<td>Expose enough concurrency, but not too much</td>
</tr>
<tr>
<td>Orchestration</td>
<td>Yes</td>
<td>Balance workload</td>
</tr>
<tr>
<td>Orchestration</td>
<td>Yes</td>
<td>Reduce non-inherent communication via data locality</td>
</tr>
<tr>
<td>Mapping</td>
<td>Yes</td>
<td>Reduce communication and synchronization cost (as seen by processor)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Schedule tasks to satisfy dependencies early</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exploit locality in network topology</td>
</tr>
</tbody>
</table>

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.

Figure 1: 2D Grid for Equation Solver.
Equation Solver: Initialization Code

```c
1 int n; /* size of matrix: (n+2 by n+2) elements */
2 float **A, *diff = 0;
3 main()
4 begin
5     read(n); /* read input parameter: matrix size */
6     A = malloc (2D array);
7     initialize(A); /* initialize the matrix A */
8     Solve(A); /* call routine to solve the equation */
9 end main
```

Equation Solver: Loop Code

```c
1 procedure Solve(A) /* solve the equation system */
2 begin
3     int i, j, done = 0;
4     float temp;
5     while (!done) do /* outermost loop over sweeps */
6         diff = 0; /* initialize maximum difference to 0 */
7             for i = 1 to n do /* sweep over non-border points of grid */
8                 for j = 1 to n do /* sweep over non-border points of grid */
9                     temp = A[i][j];/* save old value of element */
11                    diff += abs(A[i][j] - temp);
12             end for
13             if (diff/(n*n) < TOL) then done = 1;
14         end for
15     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 \(\left\lfloor \frac{i}{p} \right\rfloor\)).

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.

**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
Data Parallel Model

Concept: Parallel threads perform same action on different elements of a data set and simultaneously engage in global communication at intervals.

Analogy: Imagine a company where every employee has an identical job description and title. They work independently at their own desks/cubicles, and have regularly scheduled meetings where all employees attend and give updates on their progress.

Equation Solver: Initialization Code for Data Parallel Orchestration

```c
int n, nprocs; /* size of matrix: (n+2) by (n+2) elements */
float **A; /* A is (n+2) by (n+2) array */

begin
int i, j, done = 0;
float mydiff = 0, temp;
DECOMP A[2D BLOCK * nprocs];
while ((done) do /* external loop over sweeps */
  mydiff = 0; /* initialize maximum difference to 0 */
  for all i = 1 to n do /* sweep over non-border points of grid */
    mydiff += abs(A[i][j] - temp);
  end for all
end while
end procedure
```

Equation Solver: Loop Code for Data Parallel Orchestration

```c
procedure Solve() /* solve the equation system */
begin
int i, j, done = 0;
float mydiff = 0, temp;
DECOMP A[2D BLOCK * nprocs];
while ((done) do /* external loop over sweeps */
  mydiff = 0; /* initialize maximum difference to 0 */
  for all i = 1 to n do /* sweep over non-border points of grid */
    temp = A[i][j]; /* save old value of element */
    mydiff += abs(A[i][j] - temp);
  end for all
end while
if (mydiff < TOL) then done = 1;
end while
end procedure
```

Equation Solver: Comments on Data Parallel Orchestration

This solver lends itself well to a data parallel implementation. Computational structure is simple, regular, and well-defined. May not be feasible if the computations are irregular and less well-behaved.
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; /* 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 */
15.         mydi += 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: 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 */
15.         mydi += 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.
Message Passing Model

Concept: Parallel threads have private address space, but communicate via explicit exchange via send/receive operations.

Analogy: In this company people work in their own offices/cubicles, but they exchange phone calls, e-mails, and instant messages from time to time.

Equation Solver: Initialization Code for Message Passing

```c
1 int n, nprocs; /* size of matrix: (n+2 by n+2) elements */
2 float **A; /* A is (n+2) by (n+2) array */
3 main()
4 begin
5 read(n);
6 read(nprocs);
7 create(nprocs-1, Solve, A); /* call routine to solve the equation */
8 while (done) do /* outermost loop over sweeps */
9 mydiff = 0; /* initialize maximum difference to 0 */
10 if (pid != 0) then send(A[0,0], n*sizeof(float), pid-1, ROW);
11 if (pid != nprocs-1) then send(A[n'+1,0], n*sizeof(float), pid+1, ROW);
12 if (pid != nprocs-1) then receive(A[0,0], n*sizeof(float), pid+1, ROW);
13 for i = 1 to n do /* for i = minm to mymax do */
14 for j = 1 to n do /* sweep over non-border points of grid */
15 temp = A[i,j]; /* save old value of element */
17 mydiff += abs(A[i,j] - temp);
18 send(done, sizeof(int), 0, DONE);
19 end for
20 end for
21 if (pid != 0) then receive(mydiff, sizeof(float), 0, DIFF);
22 else for i = 1 to nprocs do
23 receive(mydiff, sizeof(float), i, DIFF);
24 receive(done, sizeof(int), i, DONE);
25 end for
26 endif
27 end while
28 end main
```

Equation Solver: Loop Code for Message Passing

```c
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, n = n/nprocs, done = 0;
5 float temp, tempdiff, mydiff = 0;
6 myA = G_MALLOC(2D array);
7 initialize(myA); /* initialize the matrix A */
8 while (done) do /* outermost loop over sweeps */
9 mydiff = 0; /* initialize maximum difference to 0 */
10 if (pid != 0) then send(&myA[1,0], n*sizeof(float), pid-1, ROW);
11 if (pid != nprocs-1) then send(&myA[0,0], n*sizeof(float), pid+1, ROW);
12 if (pid != nprocs-1) then receive(&myA[0,0], n*sizeof(float), pid+1, ROW);
13 for i = 1 to n do /* for i = minm to mymax do */
14 for j = 1 to n do /* sweep over non-border points of grid */
15 temp = myA[i,j]; /* save old value of element */
17 mydiff += abs(myA[i,j] - temp);
18 end for
19 end for
20 if (pid != 0) then send(mydiff, sizeof(float), 0, DIFF);
21 else for i = 1 to nprocs do
22 receive(mydiff, sizeof(float), i, DIFF);
23 end for
24 receive(done, sizeof(int), 0, DONE);
25 endif
26 end while
27 end procedure
```

Equation Solver: Comments on Message Passing

Orchestration

This example uses synchronous SEND/RECEIVE for simplicity.

Asynchronous operations allow for greater overlap between computation and communication.
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.

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.