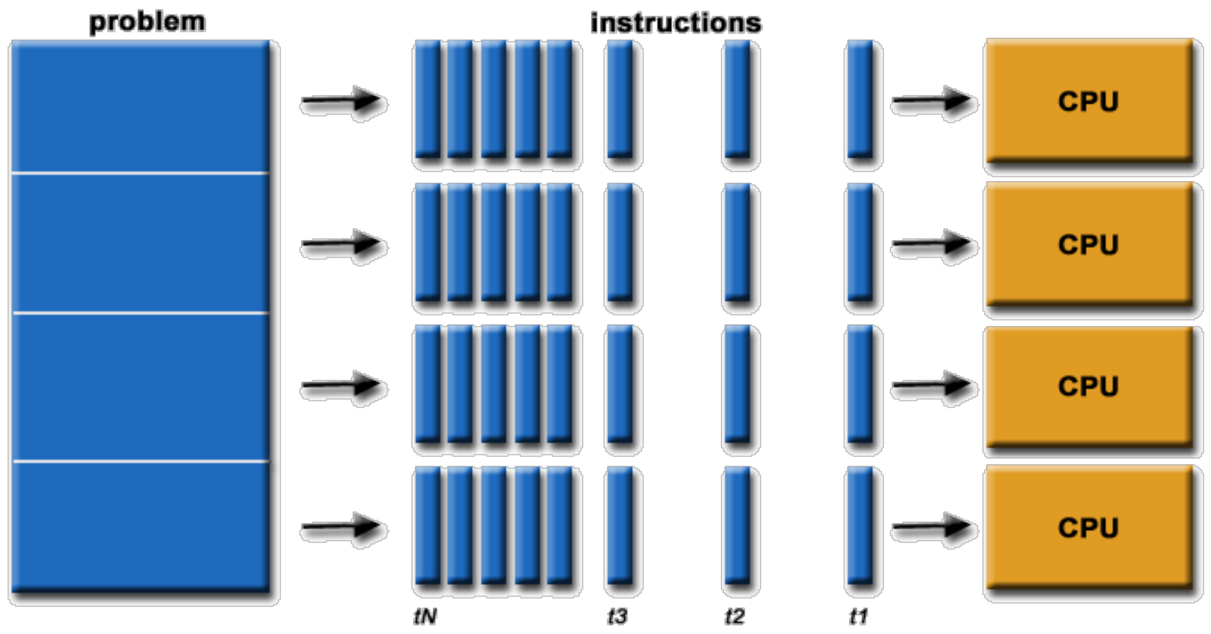
HOW TO WRITE PARALLEL PROGRAMS AND UTILIZE CLUSTERS EFFICIENTLY

Sarvani Chadalapaka
HPC Administrator
University of California Merced, Office of Information Technology
schadalapaka@ucmerced.edu
it.ucmerced.edu

WHAT IS PARALLEL COMPUTING?

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.
- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



PARALLEL COMPUTING: THE COMPUTATIONAL PROBLEM

The computational problem usually demonstrates characteristics such as the ability to be:

- Broken apart into discrete pieces of work that can be solved simultaneously
- Execute multiple program instructions at any moment in time
- Solved in less time with multiple compute resources than with a single compute resource

HOW TO WRITE PARALLEL PROGRAMS

- Undoubtedly, the first step in developing parallel software is to first understand the problem that you wish to solve in parallel. If you are starting with a serial program, this necessitates understanding the existing code also.
- Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.

Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.

This problem is able to be solved in parallel. Each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation is also a parallelizable problem.

EXAMPLE OF A NON-PARALLELIZABLE PROBLEM

Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula:

$F(k + 2) = F(k + 1) + F(k)$

This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones. The calculation of the $k + 2$ value uses those of both $k + 1$ and k. These three terms cannot be calculated independently and therefore, not in parallel.

IDENTIFY THE PROGRAM’S HOTSPOTS

- Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
- Profilers and performance analysis tools can help here
- Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

IDENTIFY BOTTLENECKS IN THE PROGRAM

- Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
- May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas

OTHER CONSIDERATIONS

- Identify inhibitors to parallelism. One common class of inhibitor is *data dependence*, as demonstrated by the Fibonacci sequence above.
- Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.

EFFICIENT PARALLEL PROGRAMMING

An efficient parallel implementation of a serial program may not be obtained by finding efficient parallelizations of each of its steps. Rather, the best parallelization may be obtained by stepping back and devising an entirely new algorithm.

Example:

Compute n values and add them together

We know that this can be done with the following serial code:

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute next value(. . .);
   sum += x;
}
```

- Now suppose we also have p cores and p is much smaller than n.
- Then each core can form a partial sum of approximately n/p values:

```
my sum = 0;
my first i = . . . ;
my last i = . . . ;
for (my i = my first i; my i < my last i; my i++)
{
   my x = Compute next value(. . .);
   my sum += my x;
}
```

For example, if there are eight cores, n = 24, and the 24 calls to **`Compute_next_value`** return the values

1, 4, 3, 9, 2, 8, 5, 1, 1, 6, 2, 7, 2, 5, 0, 4, 1, 8, 6, 5, 1, 2, 3, 9,

then the values stored in my sum might be:

Core	0	1	2	3	4	5	6	7
`my_sum`	8	19	7	15	7	13	12	14

When the cores are done computing their values of my sum, they can form a global sum by sending their results to a designated "master" core, which can add their results:

```
if (I'm the master core) {
 sum = my x;
  for each core other than myself {
   receive value from core; sum += value;
  }
} else { send my x to the master; }
```

AMDAHL’S LAW

If F is the fraction of a calculation that is sequential, and (1-F) is the fraction that can be parallelized, then the maximum speed-up that can be achieved by using P processors is 1/(F+(1-F)/P).

If 90% of a calculation can be parallelized (i.e. 10% is sequential) then the maximum speed-up which can be achieved on 5 processors is 1/(0.1+(1-0.1)/5) or roughly 3.6 (i.e. the program can theoretically run 3.6 times faster on five processors than on one)

The point that Amdahl was trying to make was that using lots of parallel processors was not a viable way of achieving the sort of speed-ups that people were looking for. i.e. it was essentially an argument in support of investing effort in making single processor systems run faster.

Proof for Traditional Problems: If the fraction of the computation that cannot be divided into concurrent tasks is *f*, and no overhead incurs when the computation is divided into concurrent parts, the time to perform the computation with *n* processors is given by $t_p \geq ft_s + [(1 - f)t_s] / n$, as shown below:

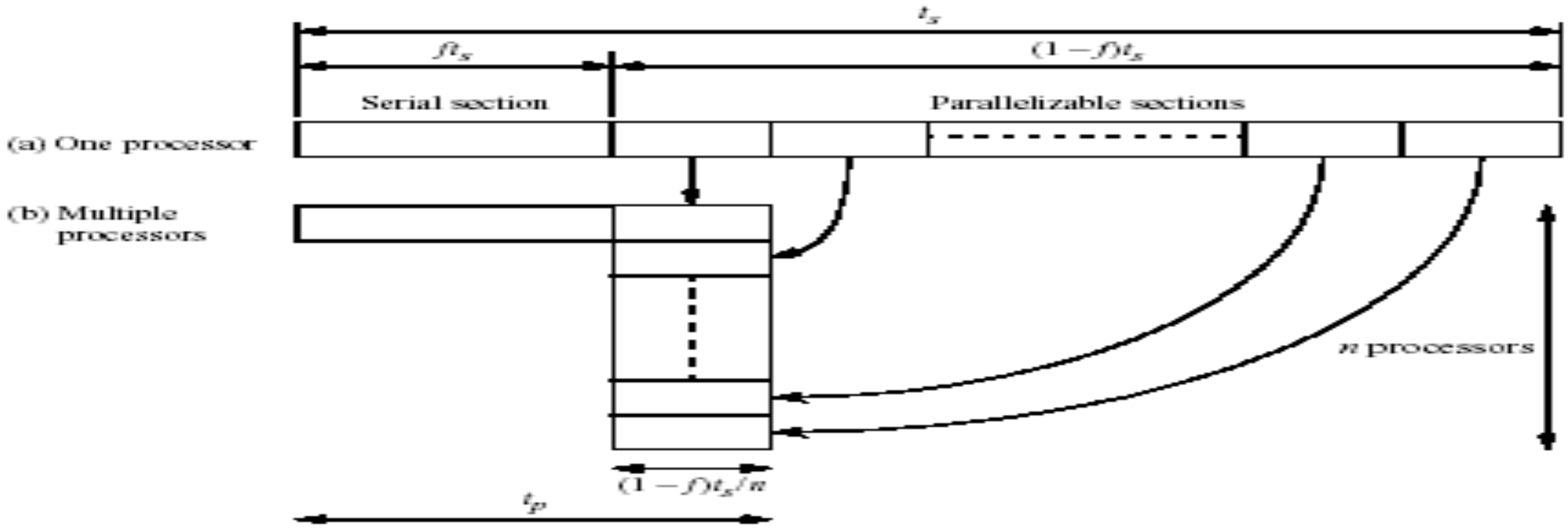


Figure 1.29 Parallelizing sequential problem — Amdahl's law.

CONSEQUENCES OF AMDAHL’S LIMITATIONS TO PARALLELISM

- For a long time, Amdahl’s law was viewed as a fatal flaw to the usefulness of parallelism.
- Amdahl’s law is valid for traditional problems and has several useful interpretations.
- Some textbooks show how Amdahl’s law can be used to increase the efficient of parallel algorithms
  - See Reference (16), Jordan & Alaghband textbook
- Amdahl’s law shows that efforts required to further reduce the fraction of the code that is sequential may pay off in large performance gains.
- Hardware that achieves even a small decrease in the percent of things executed sequentially may be considerably more efficient.

LIMITATIONS OF AMDAHL’S LAW

- A key flaw in past arguments that Amdahl’s law is a fatal limit to the future of parallelism is
  - **Gustafon’s Law:** The proportion of the computations that are sequential normally decreases as the problem size increases.
- Other limitations in applying Amdahl’s Law:
  - Its proof focuses on the steps in a particular algorithm, and does not consider that other algorithms with more parallelism may exist
- Amdahl’s law ignores the communication cost in MIMD (Multiple Instruction Multiple Data) systems.
- On communications-intensive applications, there could be an additional communication slowdown due to network congestion.
- As a result, Amdahl’s law usually overestimates speedup achievable

EFFECTIVE UTILIZATION OF MERCED CLUSTER

- Know your code, its resource needs, and resource scaling. A few minutes of planning and benchmarking will yield great returns in productivity later.
- Always use the queue system to run jobs. Do not run on the head-node.
- Don't mindlessly run calculations. Know how things are progressing and know if your calculation is running as expected. If something seems off, dig into it and understand what's going on.
- If you think calculations are not running well, ask for help from colleagues and the HPC staff.