Table of Contents

1 Parallel Performance Metrics
   - Speedup and Efficiency of Parallel Code
   - Effect of Overhead
   - Amdahl’s Law
   - Thomas timing examples - Parallel Model
   - Code Performance: Serial Looptest.f90
Total Parallel Program Time

- The total parallel program run time is a function of a large number of variables: number of processing elements (PEs); communication; hardware (cpu, memory, software, network), and the program being run (algorithm, problem size, # Tasks, complexity, data distribution); parallel libraries:

\[ T = \mathcal{F}(PEs, N, \text{Tasks}, I/O, \text{Communication}, \ldots) \]

- The execution time required to run a problem of size \( N \) on processor \( i \), is a function of the time spent in different parts of the program (computation, communication, I/O, idle):

\[ T^i = T^i_{\text{comp}} + T^i_{\text{comm}} + T^i_{\text{io}} + T^i_{\text{idle}} \]

- The total time is the sum of the times over all processes averaged over the number of the processors:

\[ T = \frac{1}{p} \left( \sum_{i=0}^{p-1} T_{\text{comp}} + \sum_{i=0}^{p-1} T_{\text{comm}} + \sum_{i=0}^{p-1} T_{\text{io}} + \sum_{i=0}^{p-1} T_{\text{idle}} \right) \]
Parallel Performance Metrics
Speedup and Efficiency of Parallel Code

Speedup

- Refers to how much faster the parallel algorithm runs than a corresponding sequential algorithm.
- $T_{ser} = $ time between when first processor begins execution to when the last processor completes its tasks.
- Speedup is defined to be:
  \[
  S_p = \frac{T_{ser}}{T_{par}}
  \]
- Where:
  - $p = $ number of cores (processors, PE’s)
  - $T_{ser} = $ serial execution time
  - $T_{par} = $ parallel execution time
- Linear speedup, or ideal speedup, is obtained when $S_p = p$:
  \[
  T_{par} = T_{ser} / p
  \]
Parallel Performance Metrics

Efficiency

- Estimation of how well the processors are used to solve the problem vs. effort is wasted in communication and synchronization.
- $T_{\text{elap}} = \text{time between when first processor begins execution to when the last processor completes its tasks}$

$$E = \frac{S}{p} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}$$

- Where:
  - $p = \text{number of cores (processors, PE’s)}$
  - $T_{\text{ser}} = \text{serial execution time}$
  - $T_{\text{par}} = \text{parallel execution time}$
- Efficiency is typically between zero and one
Test data from Pacheco IPP (Ch 2) demonstrating effect of problem size on the speedup and efficiency.

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>S</td>
<td>1.00</td>
<td>1.90</td>
<td>3.10</td>
<td>4.80</td>
<td>6.20</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1.00</td>
<td>0.95</td>
<td>0.78</td>
<td>0.60</td>
<td>0.39</td>
</tr>
<tr>
<td>Original</td>
<td>S</td>
<td>1.00</td>
<td>1.90</td>
<td>3.60</td>
<td>6.50</td>
<td>10.80</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1.00</td>
<td>0.95</td>
<td>0.90</td>
<td>0.81</td>
<td>0.68</td>
</tr>
<tr>
<td>Double</td>
<td>S</td>
<td>1.00</td>
<td>1.90</td>
<td>3.90</td>
<td>10.80</td>
<td>14.20</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1.00</td>
<td>0.95</td>
<td>0.98</td>
<td>0.94</td>
<td>0.89</td>
</tr>
</tbody>
</table>
Speedup and Efficiency of Parallel Code

Speedup vs. Number of Processors

- **Half**
- **Original**
- **Double**

![Graph showing speedup vs. number of processors for different configurations.](image-url)
Parallel Performance Metrics

Speedup and Efficiency of Parallel Code

Efficiency vs. Number of Processors

- **Efficiency** vs. **Number of Processors**

Graph showing efficiency of parallel code with different numbers of processors. The graph compares the efficiency of parallel code compared to the original and double versions, illustrating the impact of processor count on efficiency.
Parallel Performance Metrics

Speedup and Efficiency of Parallel Code

Run Times vs. Number of Processors

- **Run Time (seconds)**
  - 1.20
  - 1.00
  - 0.80
  - 0.60
  - 0.40
  - 0.20
  - 0.00

- **Number of Processors**
  - 0
  - 5
  - 10
  - 15
  - 20

- **Graph Lines**
  - **Half**
  - **Original**
  - **Double**
Effect of overhead

\[ T_{\text{parallel}} = T_{\text{serial}} / p + T_{\text{overhead}} \]
Amdahl’s Law

- Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.
Example

- We can parallelize 90% of a serial program.
- Parallelization is “perfect” regardless of the number of cores $p$ we use.
- $T_{\text{serial}} = 20$ seconds
- Runtime of parallelizable part is

$$0.9 \times T_{\text{serial}} / p = 18 / p$$
Example (cont.)

- Runtime of “unparallelizable” part is
  
  \[ 0.1 \times T_{\text{serial}} = 2 \]

- Overall parallel run-time is
  
  \[ T_{\text{parallel}} = 0.9 \times T_{\text{serial}} / p + 0.1 \times T_{\text{serial}} = 18 / p + 2 \]
Example (cont.)

- Speed up

\[ S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}} / p + 0.1 \times T_{\text{serial}}} = \frac{20}{18 / p + 2} \]
Scalability

- In general, a problem is *scalable* if it can handle ever increasing problem sizes.
- If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is *strongly scalable*.
- If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is *weakly scalable*.
# Customized Timings: Parallel Framework

## Table II.

<table>
<thead>
<tr>
<th>Section</th>
<th>Serial</th>
<th>16 Processors</th>
<th>32 Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tinit</td>
<td>48571</td>
<td>24285</td>
<td>16190</td>
</tr>
<tr>
<td>Tloop</td>
<td>59451</td>
<td>29725</td>
<td>19817</td>
</tr>
<tr>
<td>Twall</td>
<td>108083</td>
<td>54041</td>
<td>36027</td>
</tr>
</tbody>
</table>

## Table III.

<table>
<thead>
<tr>
<th>Section</th>
<th>Serial</th>
<th>16 Processors</th>
<th>32 Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tpres</td>
<td>31619</td>
<td>15810</td>
<td>10540</td>
</tr>
<tr>
<td>Tfio</td>
<td>17961</td>
<td>8981</td>
<td>5987</td>
</tr>
<tr>
<td>Tsrgs</td>
<td>3026</td>
<td>1513</td>
<td>1009</td>
</tr>
<tr>
<td>TVelw</td>
<td>1736</td>
<td>868</td>
<td>579</td>
</tr>
<tr>
<td>TVelu</td>
<td>1726</td>
<td>863</td>
<td>575</td>
</tr>
<tr>
<td>TVelv</td>
<td>1716</td>
<td>858</td>
<td>572</td>
</tr>
<tr>
<td>TbcondP</td>
<td>448</td>
<td>224</td>
<td>150</td>
</tr>
<tr>
<td>TvelcorV</td>
<td>120</td>
<td>61</td>
<td>40</td>
</tr>
<tr>
<td>TvelcorW</td>
<td>110</td>
<td>55</td>
<td>36</td>
</tr>
<tr>
<td>TvelcorU</td>
<td>109</td>
<td>54</td>
<td>367</td>
</tr>
<tr>
<td>TbcondW</td>
<td>22</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>TbcondU</td>
<td>22</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>TbcondV</td>
<td>20</td>
<td>11</td>
<td>67</td>
</tr>
<tr>
<td>Tloop (meas)</td>
<td>58635</td>
<td>29317</td>
<td>19545</td>
</tr>
</tbody>
</table>
Runtime vs Number of Iterations as a Function of the Number of Processors (linear-linear)

- SerRef
- Par4
- Par8
- Par16
- Par32
Runtime vs Number of Iterations as a Function of the Number of Processors (log-log)
Runtime vs Number of Iterations as a Function of the Number of Processors (log-linear)
Parallel Performance Metrics

Speedup (Ts/Tp)

- **Speedup**
- **Ideal**

Processor Elements

Speed
Parallel Performance Metrics

Thomas timing examples - Parallel Model

**Efficiency** = \((T_s/T_p)/P = T_s/(P \cdot T_p)\)

Graph showing efficiency as a function of processors, with lines indicating efficiency and ideal cases.
## Parallel Performance Metrics

**Thomas timing examples - Parallel Model**

```c
/* hello.c by James Otto, 1/31/11
-- for running serial processes
    on a cluster... see batch.hello */
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    char cptr[100];
    gethostname(cptr,100);
    printf("Hello, world from %s\n", cptr);
    return 0;
}

---

**COMPILE & RUN SERIAL PGM**

[tuckoo]$ mpicc -o hello hello.c
[mthomas@tuckoo ex.2014]$ mpirun -np 5 ./hello
Hello, world from tuckoo
Hello, world from tuckoo
Hello, world from tuckoo
Hello, world from tuckoo
Hello, world from tuckoo

---

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "mpi.h"
int main (int argc, char* argv[])
{
    int rank, nprocs, ierr, i, error=0;
    MPI_Status status;
    ierr = MPI_Init(&argc, &argv);
    if (ierr != MPI_SUCCESS) {
        printf("MPI initialization error\n");
    }
    // processing element ID
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // ID of communicator connecting PE's
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    printf("Hello Processor: rank: %d, nprocs: %d\n", rank, nprocs);
    MPI_Finalize();
    return 0;
}

---

**COMPILE & RUN PARALLEL PGM**

[tuckoo]$ mpicc -o hello_mpi hello_mpi.c
[tuckoo]$ mpirun -np 5 ./hello_mpi
Hello Processor: rank: 0, nprocs: 5
Hello Processor: rank: 1, nprocs: 5
Hello Processor: rank: 3, nprocs: 5
Hello Processor: rank: 4, nprocs: 5
Hello Processor: rank: 2, nprocs: 5
program looptest
    implicit none
    integer, parameter :: max=10000
    integer :: i, j
    double precision :: tws, tte, ts, te
    double precision :: a(max, max), x(max), y(max)
    call cpu_time(tws)
    ! initialize arrays
    a=0.0; x=0.0; y=0.0
    do i=1, max
        x(i) = i
        y(i) = max-i
        do j=1, max
            a(i, j) = 10*j + i
        enddo
    enddo
    ! compute loop 1
    call cpu_time(ts)
    call loop1(y, max)
    call cpu_time(te)
    print *, "Telap: loop 1 = ", (te - ts)
    ! compute loop 2
    ts=0.0; te=0.0;
    call cpu_time(ts)
    call loop2(y, max)
    call cpu_time(te)
    print *, "Telap: loop 2 = ", (te - ts)
    ! compute loop 3
    ts=0.0; te=0.0;
    call cpu_time(ts)
    call loop3(y, max)
    call cpu_time(te)
    print *, "Telap: loop 3 = ", (te - ts)
    call cpu_time(twe)
    print *, "Wallclock Time: ", (twe - tws)
Code Example: run job from command line

```
[mthomas@tuckoo]$ cat makefile

MAKE FILE

MPIF90 = mpif90
MPICC = mpicc
CC = gcc

all: looptst looptstp
looptst: looptst.f90
$(MPIF90) -o looptst looptst.f90

looptstp: looptst.f90
$(MPIF90) -p -o looptstp looptst.f90

clean:
rm -rf *.o looptst looptst mpi

[mthomas@tuckoo]$ make

SERIAL JOB: FROM COMMAND LINE

Testing FORTRAN loops (column major):
Telap: loop 1 = 960.8539 msec
Telap: loop 2 = 580.9109 msec
Telap: loop 3 = 1744.7349 msec
Wallclock Time: = 5861.1099 msec

PROFILING: using -p option in make

[mthomas@tuckoo]$ gprof looptstp gmon.out

Flat profile:

<table>
<thead>
<tr>
<th>cumulative</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
</tr>
<tr>
<td>37.58</td>
<td>1.39</td>
<td>1.39</td>
</tr>
<tr>
<td>27.04</td>
<td>2.40</td>
<td>1.00</td>
</tr>
<tr>
<td>23.25</td>
<td>3.26</td>
<td>0.86</td>
</tr>
<tr>
<td>10.95</td>
<td>3.67</td>
<td>0.41</td>
</tr>
</tbody>
</table>
```

```
Run Serial Job In Queue

---

= SUBMIT SERIAL JOB TO QUEUE

[mthomas@tuckoo looptst]$ cat batch.looptst
#!/bin/sh
#PBS -V
#PBS -l nodes=2:ppn=4:core4
#PBS -N looptstp
#PBS -j oe
#PBS -q batch
cd $PBS_O_WORKDIR
NCORES=`wc -w < $PBS_NODEFILE`
echo "looptstp—test using $NCORES cores..."
mpirun -np 4 -hostfile $PBS_NODEFILE
    --nooversubscribe ./looptstp

[mthomas@tuckoo looptst]$ !qsub
qsub batch.mpi—looptstp
16478.tuckoo.sdsu.edu

---

= OUTPUT (asynchronous)

| Telap: loop 1 | 0.8308729 |
| Telap: loop 1 | 0.8308739 |
| Telap: loop 1 | 0.8328739 |
| Telap: loop 1 | 0.8428719 |

| Telap: loop 2 | 0.4499310 |
| Telap: loop 2 | 0.4549309 |
| Telap: loop 2 | 0.4559310 |
| Telap: loop 2 | 0.4559310 |

| Telap: loop 3 | 0.9898489 |
| Telap: loop 3 | 0.9908489 |
| Telap: loop 3 | 0.9918490 |
| Telap: loop 3 | 1.0078469 |

| Wallclock Time: | 5.02523599 |
| Wallclock Time: | 5.0262349 |
| Wallclock Time: | 5.02823599 |
| Wallclock Time: | 5.049231 |

Note: no gain by using multiple PE’s —— > no MPI calls in code
Add MPI Calls

program looptest
!
implicit none
include "mpif.h"
integer, parameter :: max=10000
double precision, allocatable :: a(:,,:), x(:,), y(:)
double precision :: tws, tte, ts, te
integer :: i, j, rank, nprocs, ierr, token
integer :: status(MPI_STATUS_SIZE)

call cpu_time(tws)
call MPI_INIT(ierr)
if (ierr.ne. MPI_SUCCESS) then
   print *, "Error: initing in MPI_INIT()"
   stop
endif

! find out how many processes \\ &
local process rank
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)

maxloc=gl_max/nprocs
allocate(a(maxloc,maxloc), x(maxloc), &
   y(maxloc), stat=ierr)

! initialize arrays
do i=1,max
   x(i) = i; y(i) = max-i
   do j=1,max
      a(i,j) = 10*j + i
   enddo
enddo

! compute loop1
   call cpu_time(ts)
call loop1(y,max)
call cpu_time(te)
write( )

! compute loop2
   ts=0.0; te=0.0;
call cpu_time(ts)
call loop2(y,max)
call cpu_time(te)
write( )

! compute loop3
   ts=0.0; te=0.0;
call cpu_time(ts)
call loop3(y,max)
call cpu_time(te)
write( )
call cpu_time(twe)
write( )
call MPI_FINALIZE(ierr)

contains
   ..
Run MPI Job In Queue

SUBMIT JOB TO QUEUE

[mthomas@tuckoo looptst] $ !qsub
qsub batch.mpi-looptstp
16478.tuckoo.sdsu.edu

OUTPUT (asynchronous)

[mthomas@tuckoo looptst] $ cat mpi-looptstp.o16485
mpi-looptstp-test using 8 cores...
LocaMAX: 2500
LocaMAX: 2500
LocaMAX: 2500

PE[ 0]: Telap, loop 1= 0.07698800
PE[ 1]: Telap, loop 1= 0.07698800
PE[ 2]: Telap, loop 1= 0.07598900
PE[ 3]: Telap, loop 1= 0.07698800

PE[ 0]: Telap, loop 2= 0.03799400
PE[ 1]: Telap, loop 2= 0.03799500
PE[ 2]: Telap, loop 2= 0.03799400
PE[ 3]: Telap, loop 2= 0.03699500

PE[ 0]: Telap, loop 3= 0.07898800
PE[ 1]: Telap, loop 3= 0.07998700
PE[ 2]: Telap, loop 3= 0.07998800
PE[ 3]: Telap, loop 3= 0.08098700

PE[ 0]: Telap, Twall= 0.33594800
PE[ 1]: Telap, Twall= 0.33794800
PE[ 2]: Telap, Twall= 0.33294900
PE[ 3]: Telap, Twall= 0.34294600

Note: \( T_{wall} \) reduced from 5+ seconds to 0.3
mpi-looptst RunTime (Twall)

Note: Ideal runtime computed using $T_{ideal} = \frac{T_{ser}}{p}$
mpi-looptst: Speedup

![Graph showing the speedup for mpi-looptst with Nprocs on the x-axis and Speedup on the y-axis. The graph compares the actual speedup to the ideal speedup.]