Introduction to Parallel Computing

damien.francois@uclouvain.be
October 2015
Why parallel?

Speed up – Solve a problem faster
  → more processing power
    (a.k.a. strong scaling)

Scale up – Solve a larger problem
  → more memory and network capacity
    (a.k.a. weak scaling)

Scale out – Solve many problems
  → more storage capacity
Agenda

1. General concepts
2. Hardware
3. Programming models
4. User tools
General concepts
Amdahl's Law

https://en.wikipedia.org/wiki/Amdahl%27s_law
“Scaling can be linear anyway providing sufficiently enough data are used.”
Parallel overhead

Example of Parallel Communications Overhead and Complexity: actual callgraph from the simple parallel "hello world" program shown. Most of the routines are from communications libraries.

void main(int argc, char *argv[])
{
    int myrank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello World\n", myrank, size);
    MPI_Finalize();
}
How parallel?

- Parallelization means
  - **distributing work** to processors
  - **distributing data** (if memory is distributed)
- and
  - **synchronization** of the distributed work
  - **communication** of remote data to local processor (if memory is distributed)
Decomposition

- **Work decomposition**: task-level parallelism
- **Data decomposition**: data-level parallelism
- **Domain decomposition**: decomposition of work and data is done in a higher model, e.g. in the reality
Collaboration

- **Synchronous** (SIMD) at the processor level
- **Fine-grained** parallelism if subtasks must communicate many times per second (instruction level); loosely synchronous
- **Coarse-grained** parallelism if they do not communicate many times per second (function-call level)
- **Embarrassingly parallel** if they rarely or never have to communicate (asynchronous)
Parallel programming paradigms

- **Task-farming** (master/slave or work stealing)
- **Pipelining** (A->B->C, one process per task concurrently)
- **SPMD** (pre-defined number of processes created)
- **Divide and Conquer** (processes spawned at need and report their result to the parent)
- **Speculative parallelism** (processes spawned and result possibly discarded)
Hardware
At the processor level

- Instruction-level parallelism (ILP)
  - Instruction pipelining
  - Superscalar execution
  - Out-of-order execution
  - Speculative execution
- Single Instruction Multiple Data (SIMD)
At the computer level

- Multithreading
  - SMP
  - NUMA
- Accelerators
At the system level

Distributed computing – Grids – Clusters – Cloud
Distributed computing

Folding@home

Where are the Folding@home donors located?
Update: May 2008 FAH donor world map

We've updated our map by hand in May 2008. Click on the map to get a larger version. We're still looking for ways to reliably automate this process. Thanks to Geostats for help with this map.
Cluster computing
Grid computing

"Currently WLCG is made up of more than 170 computing centers in 36 countries... The WLCG is now the world's largest computing grid"
Cloud computing

Google Compute Engine

Run your large-scale computing workloads on Linux virtual machines hosted on Google's infrastructure. Sign up to request access.

Use Cases (Initial)

- Batch processing: Execute batch processing jobs like video transcoding and rendering on Google Compute Engine.
- Data processing: Analyze massive amounts of data in the cloud using frameworks like Hadoop.
Programming models
Programming models

- CUDA, OpenCL
- PThreads, OpenMP, TBB
- MPI
- CoArray, UPC
- MapReduce
- Boinc
CUDA, OpenCL (own session)

- Shared memory
- Single Machine / Multiple devices
- Work decomposition
- Single Program Multiple Data
- Synchronous (SIMD)
CUDA (C/C++/Fortran)

```c
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void hello(char *a, int *b) {
    a[threadIdx.x] += b[threadIdx.x];
}

int main() {
    char a[N] = "Hello \0\0\0\0\0\0\0\0";
    int b[N] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    char *ad;
    int *bd;
    const int csize = sizeof(char);
    const int isize = sizeof(int);
    printf("%s", a);
    cudaMalloc((void**)&ad, csize);
    cudaMalloc((void**)&bd, isize);
    cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
    cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );
    dim3 dimBlock( blocksize, 1 );
    dim3 dimGrid( 1, 1 );
    hello<<dimGrid, dimBlock>>>(ad, bd);
    cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );
    cudaFree( ad );
    cudaFree( bd );
    printf("%s\n", a);
    return EXIT_SUCCESS;
}
```
OpenCL (C/C++)

Pthreads, OpenMP (own session)

- Shared memory
- Single machine
- Work decomposition
- Single Program Multiple Data
- Fine-grained/Coarse-grained parallelism
Pthreads

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS 5

void *perform_work(void *argument) {
    int passed_in_value;
    passed_in_value = *((int *) argument);
    printf("Hello World from thread %d\n", passed_in_value);
    /* optionally: insert more useful stuff here */
    return NULL;
}

int main(void) {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    int result_code, index;

    // create all threads one by one
    for (index = 0; index < NUM_THREADS; ++index) {
        thread_args[index] = index;
        result_code = pthread_create(&threads[index], NULL, perform_work, (void *) &thread_args[index]);
        assert(0 == result_code);
    }

    // wait for each thread to complete
    for (index = 0; index < NUM_THREADS; ++index) {
        // block until thread 'index' completes
        result_code = pthread_join(threads[index], NULL);
        assert(0 == result_code);
    }

    printf("In main: All threads completed successfully\n");
    exit(EXIT_SUCCESS);
}
```

https://en.wikipedia.org/wiki/POSIX_Threads
OpenMP (C/Fortran)

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
    int th_id, nthreads;
    #pragma omp parallel private(th_id)
    {
        th_id = omp_get_thread_num();
        printf("Hello World from thread %d\n", th_id);
        #pragma omp barrier
        if (th_id == 0) {
            nthreads = omp_get_num_threads();
        }
    }
    return EXIT_SUCCESS;
}
```

https://en.wikipedia.org/wiki/OpenMP
Threading Building Blocks

- Shared memory
- Single machine
- Data decomposition, Work decomposition
- Single Program Multiple Data
- Fine-grained parallelism
Threading Building Blocks (C++)

```c++
#include "tbb/tbb.h"
#include <iostream>
using namespace tbb;
using namespace std;

class say_hello() {
    const char* message;

public:
    say_hello(const char* str) : message(str) { }
    void operator()() const {
        cout << message << endl;
    }
};

int main() {
    task_group tg;
    tg.run(say_hello("child 1")); // spawn task and return
    tg.run(say_hello("child 2")); // spawn another task and return
    tg.wait(); // wait for tasks to complete
}
```
Message Passing Interface
(own session)

- Distributed memory
- Multi-machine (Cluster computing)
- Data decomposition, Work decomposition
- Single Program Multiple Data
- Coarse-grained parallelism
MPI (C/Fortran)

```c
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[]) {
  char idstr[32];
  char buff[BUFSIZE];
  int numprocs;
  int myid;
  int i;
  MPI_Status stat;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if (myid == 0) {
    printf("%d: We have %d processors\n", myid, numprocs);
    for (i = 1; i < numprocs; i++)
      printf("%d: %s\n", myid, buff);
    for (i = 1; i < numprocs; i++)
      MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
    printf("%d: %s\n", myid, buff);
  }
  else
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    printf("%d: Processor %s\n", myid);
  strcat(buff, idstr, BUFSIZE - 1);
  strcat(buff, "reporting for duty", BUFSIZE - 1);
  MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
}

MPI_Finalize();
return 0;
}
```

https://en.wikipedia.org/wiki/Message_Passing_Interface
Partitioned global address space

- Shared memory
- Multi-machine (Cluster computing)
- Data decomposition, Work decomposition
- Single Program Multiple Data
- Coarse-grained parallelism
Co-Array Fortran

```fortran
!Hello_World
implicit none
integer :: i ! Local variable
character(len=20) :: name[*] ! scalar coarray, one "name" for each image.
! Note: "name" is the local variable while "name[*-index-1]" accesses the
! variable in a specific image; "name(this_image())" is the same as "name".

! Interact with the user on Image 1; execution for all others pass by.
if (this_image() eq 1) then
    write(*,'(a,advance='no')') 'Enter your name: '
    read(*,'(a)') name

    ! Distribute information to other images
do i = 2, num_images()
    name[i] = name
    end do
end if

sync all ! Barrier to make sure the data have arrived.
!
! I/O from all images, executing in any order, but each record written is intact.
write(*,'(3a,i0)') 'Hello ',trim(name),' from image ', this_image()
end program Hello_world
```

https://en.wikipedia.org/wiki/Coarray_Fortran
UPC (C)

```c
#include <upc.h>

main()
{
    // one can use THREADS and MYTHREAD like ints
    if(MYTHREAD % 2)
        printf("Hello world: I am thread %d and I am even.\n", MYTHREAD);
    else
        printf("Hello world: I am thread %d and I am odd.\n", MYTHREAD);
}
```
MapReduce, Boinc

- Distributed memory
- Multi-machine (Distributed computing, Cloud computing)
- Data decomposition
- Divide and Conquer
- Embarrassingly parallel
Map/Reduce (Java, Python, R, ...)

```
from disco.core import Job, result_iterator

def map(line, params):
    for word in line.split():
        yield word, 1

def reduce(iter, params):
    from disco.util import kvgroup
    for word, counts in kvgroup(sorted(iter)):
        yield word, sum(counts)

if __name__ == '__main__':
    input = ['http://discoproject.org/media/text/chekhov.txt']
    job = Job().run(input=input, map=map, reduce=reduce)
    for word, count in result_iterator(job.wait()):
        print word, count
```
```c
#include "boinc_api.h"
#include "boinc.h" // boinc_init()
#include "filesys.h" // boinc_fopen(), etc...
#include "str_util.h" // boinc_close()
#include "version.h" // for parse_command_line()

int main(int argc, char **argv) {
    int rc;
    char resolved_name[512]; // physical file name for out.txt
    FILE* f;
        // file pointer for out.txt

    boinc_init_diagnostics(BOINC_DIAG_REDIRECTSTDERR|BOINC_DIAG_MEMORYLEAKCHECKENABLED|BOINC_DIAG_DUMPCALLSTACK);
    rc = boinc_init();
    if (rc){
        fprintf(stderr, "APP: boinc_init() failed. rc=%d\n", rc);
        fflush(stderr);
        exit(rc);
    }
    rc = boinc_resolve_filename("out.txt", resolved_name, sizeof(resolved_name));
    if (rc){
        fprintf(stderr, "APP: cannot resolve output file name. RC=%d\n", rc);
        boinc_finish(rc); /* back to BOINC core */
    }
    f = boinc_fopen(resolved_name, "a");
    fprintf(f, "Hello, BOINC World!\n");
    { int i, num, N;
        N = 123456789;
        fprintf(f, "Starting some computation...\n");
        for (i=0; i<N; i++){
            num = rand() + rand(); // just do something to spin the wheels
            fprintf(f, "Computation completed.\n");
        }
    }
    fclose(f);
    boinc_finish(); /* does not return */
}
```
User tools
when you cannot modify the program
Parallel processes in Bash

```
#!/bin/bash

# Usage:
# ./lower.sh [input_file [output_file]]
# Make ACTG chars lower case with extra processing.
# If output_file is not defined, stdout is used
# If input_file and output_file are not defined, stdin and stdout are used.

while read line; do
  sleep 1
  echo $line | tr ACTG actg >> ${2:-/dev/stdout}
done < ${1:-/dev/stdin}
```

```
dfr@hmem00:~/parcomp $ cat lower.sh
!/<bin/bash
#
#
# Usage:
# ./lower.sh [input_file [output_file]]
# Make ACTG chars lower case with extra processing.
# If output_file is not defined, stdout is used
# If input_file and output_file are not defined, stdin and stdout are used.

while read line; do
  sleep 1
  echo $line | tr ACTG actg >> ${2:-/dev/stdout}
done < ${1:-/dev/stdin}

dfr@hmem00:~/parcomp $ cat d.txt
G
C
A
G
dfr@hmem00:~/parcomp $ ./lower.sh d.txt
g
c
A
dfr@hmem00:~/parcomp $ ```

```
Parallel processes in Bash

```bash
# Foreground: commands end with ';
# Background, in parallel: commands end with '&' and 'wait' necessary
```
One program and many files
Several programs and one file
One program and one large file

```bash
dfr@hmem00:~/parcomp $ # One process to process the whole file
dfr@hmem00:~/parcomp $ time { cat d.txt | ./lower.sh > res.txt ; }
real   0m4.014s
user   0m0.003s
sys    0m0.009s
dfr@hmem00:~/parcomp $ # Four processes handling one line in round robin fashion
dfr@hmem00:~/parcomp $ time { cat d.txt | split --unbuffered --number r/4 --filter ./lower.sh > res.txt ; }
real   0m1.011s
user   0m0.009s
sys    0m0.021s
dfr@hmem00:~/parcomp $ !! & top -u dfr -bnl | grep lower
time { cat d.txt | split --unbuffered --number r/4 --filter ./lower.sh > res.txt ; } & top -u dfr -bnl | grep lower
[1] 12817
12822 dfr  20  0 103m 1252 1052 $  0.0  0.0  0:00.00 lower.sh
12823 dfr  20  0 103m 1252 1052 $  0.0  0.0  0:00.00 lower.sh
12824 dfr  20  0 103m 1252 1052 $  0.0  0.0  0:00.00 lower.sh
12825 dfr  20  0 103m 1252 1052 $  0.0  0.0  0:00.00 lower.sh
dfr@hmem00:~/parcomp $
real   0m1.011s
user   0m0.011s
sys    0m0.019s
[1]+ Done            time { cat d.txt | split --unbuffered --number r/4 --filter ./lower.sh > res.txt; }
dfr@hmem00:~/parcomp $```

Need recent version of Coreutils/8.22-goolf-1.4.10
Several programs and many files

```
# Sample Makefile to process each file with
# lower.sh then upper.sh
#
# all:  d1.res d2.res d3.res d4.res
#
# Build intermediary files
%.tmp:  %.txt
   ./lower.sh $< $@

# Build final result
%.res:  %.tmp
   ./upper.sh $< $@
```

"Makefile" 14L. 219C written
Several programs and many files
Summary

• You have either
  - one very large file to process
    • with one program (split – task farming)
    • with several programs (pipes, fifo – pipeline)
  - many files to process
    • with one program (xargs – task farming)
    • with many programs (make – divide and conquer)
• Always embarrassingly parallel, work decomposition
GNU Parallel

- Syntax: `parallel command :::: argument list`
GNU Parallel

- Syntax: `{}` as argument placeholder. Can be modified
GNU Parallel

- Syntax: --xapply
GNU Parallel

- Syntax: `::: arguments in file`
GNU Parallel

- Syntax: --pipe
Other interesting options

-S Use remote servers through SSH
-j n Run n jobs in parallel
-k Keep same order
--delay n Ensure there are n seconds between each start
--timeout n Kill task after n seconds if still running

Exercises

- Can you reproduce the examples with ./lower and ./upper.sh using GNU Parallel?
Solutions

- One program and many files
  
  dfr@hmem00:~/parcomp $ time -k parallel ./lower.sh {} > res.txt ::: d?.txt

- One program and one large file
  
  dfr@hmem00:~/parcomp $ time cat d.txt | parallel -k -N1 --pipe ./lower.sh {} > res.txt

- Several programs and several files
  
  dfr@hmem00:~/parcomp $ time { parallel ./lower.sh {} {}.tmp ::: d?.txt ; parallel ./upper.sh {} {}.res ::: d?.tmp ; }