Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Announcements

Announcements

- No lectures on October 22nd and October 23rd
- Github repo with the examples shown in class: *https://github.com/danieledesensi/multicore-programming/*
- **(rank - 1) % size** vs. **(rank - 1 + size) % size**

Recap

Recap

- Collectives caveats and matching rules
- Reduce, broadcast, allreduce
- pi estimation example

Performance Evaluation

Elapsed parallel time

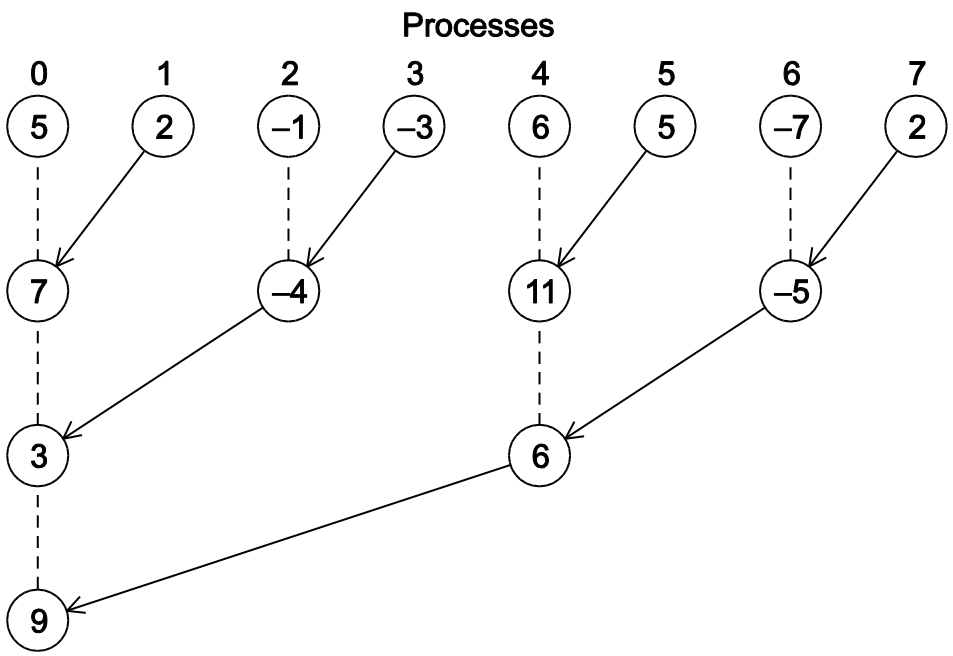
```
double MPI_Wtime(void);
```

Returns the number of seconds that have elapsed since some time in the past.

```
double start, finish;
. . .
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"
      my_rank, finish-start);
```

Which rank?

Each rank might finish at a different time. Why? Examples?



e.g., rank 0 is going to finish much later than rank 7

Solution: Report the maximum time across the ranks

Which rank?

Solution: Report the maximum time across the ranks
How?

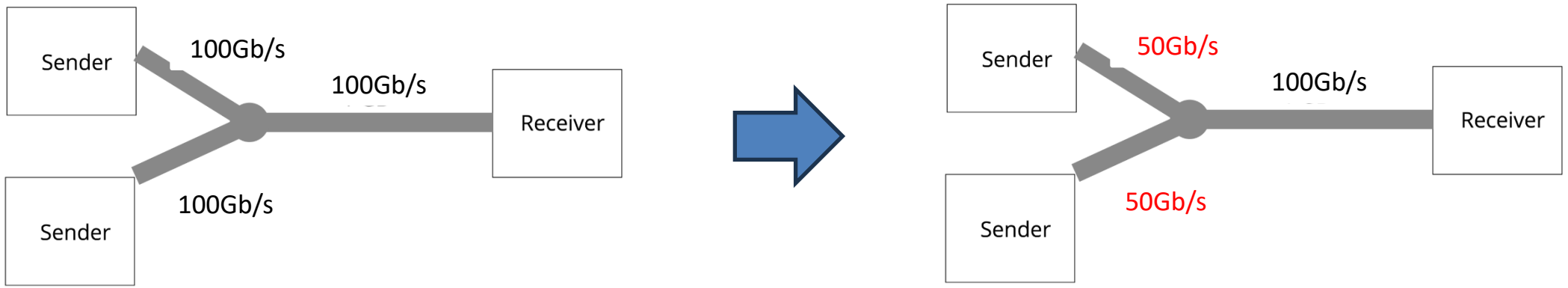
```
double local_start, local_finish, local_elapsed, elapsed;
. . .
local_start = MPI_Wtime();
/* Code to be timed */
. . .
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
   MPI_MAX, 0, comm);

if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

Is every rank going to start at the same time?

- Not necessarily
- If not, the time we report might be longer not because the application was performing poorly, but rather because someone started later than someone else
- How to ensure they are going to start at the same time?
- MPI_Barrier
- **Question:** But wait … MPI_Barrier is itself a collective, what if it is implemented with a tree? Every rank could exit the barrier at a different time
- **Answer:** Guaranteeing that they start exactly at the same time is a bit more complicated. For the purposes of this course, MPI_Barrier provides a reasonable approximation

Is one run/measurement enough?

- No, performance data is non deterministic.
- I.e., if you run your application 100 times, you will get 100 different runtimes (this is also known as ***noise***).
- **Why?**
  - On a given compute node, interference from other applications and/or operating system (context switches, cache pollution, etc…)
  - Across multiple nodes, interference on the network (is a resource shared among multiple nodes and applications)



Is one run/measurement enough?

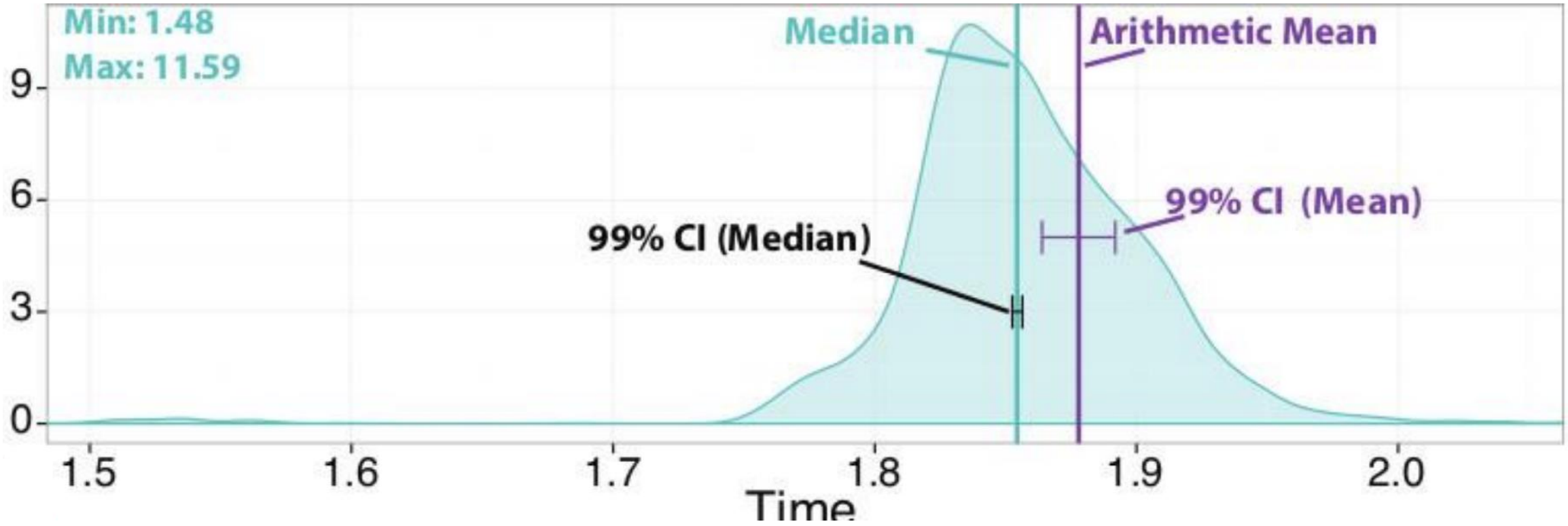
- No, performance data is non deterministic.
- **Solution:** Run the application multiple times.
- What do we report? Minimum time, maximum time, average?

> ~~Since this interaction will almost certainly not make the program run faster than it would run on a "quiet" system, we usually report the *minimum* run-time, rather than the mean or median. (For further discussion of this, see [6].)~~

- **Solution**: Report the entire distribution of timings

Is one run/measurement enough?

- **Solution**: Report the entire distribution of timings



Recap: How to take timings

- A barrier at the beginning of the application, to be sure everyone starts at the same time
- Report the maximum runtime across the ranks
- Execute your application multiple times and report the distribution of timings

Extra resources

Impact of noise/variability on applications performance:

- *"Characterizing the Influence of System Noise on Large-Scale Applications by Simulation"*
- *"Noise in the Clouds: Influence of Network Performance Variability on Application Scalability"*
- *"Mitigating Network Noise on Dragonfly Networks Through Application-aware Routing"*

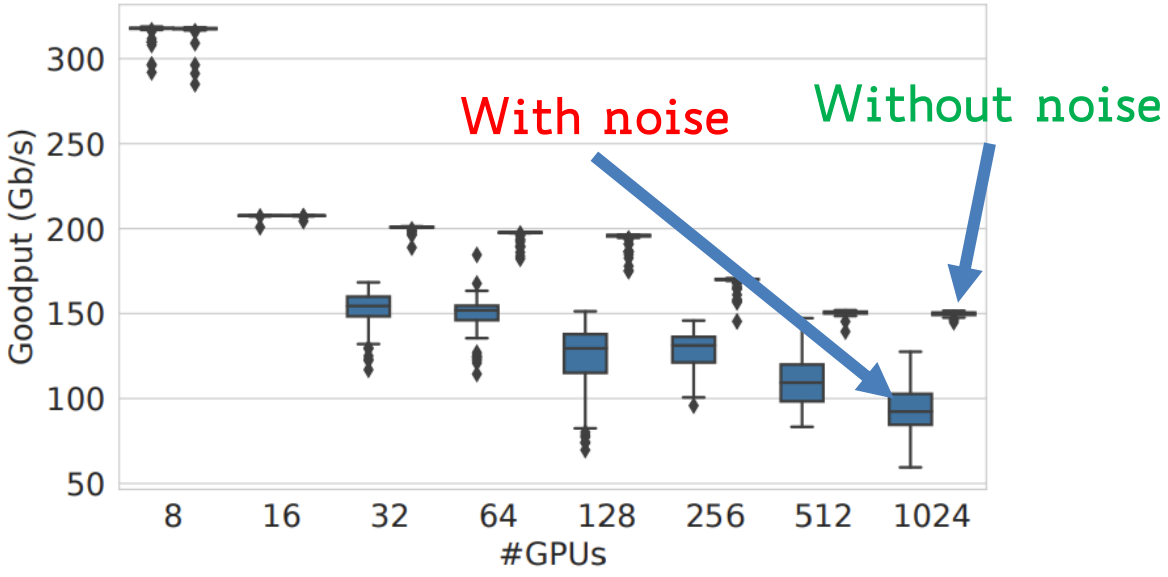
Additional things to take care of when taking timings:

- *"Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results"*
- *"Benchmarking data science: Twelve ways to lie with statistics and performance on parallel computers."*

How to plot/show data/distributions:

- https://github.com/cxli233/FriendsDontLetFriends

What's the impact of noise in practice?



(b) Allreduce

- Why does it get worst when increasing the number of ranks/GPUs?
- Intuitively: The more ranks you have, the more likely it is that at least one of them is affected by noise

"Exploring GPU-to-GPU Communication: Insights into Supercomputer Interconnects"

Questions?

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

The runtime increases with the problem size

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

The runtime decreases with the number of processes

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

Why we measure the same runtime with 8 and 16 processes?

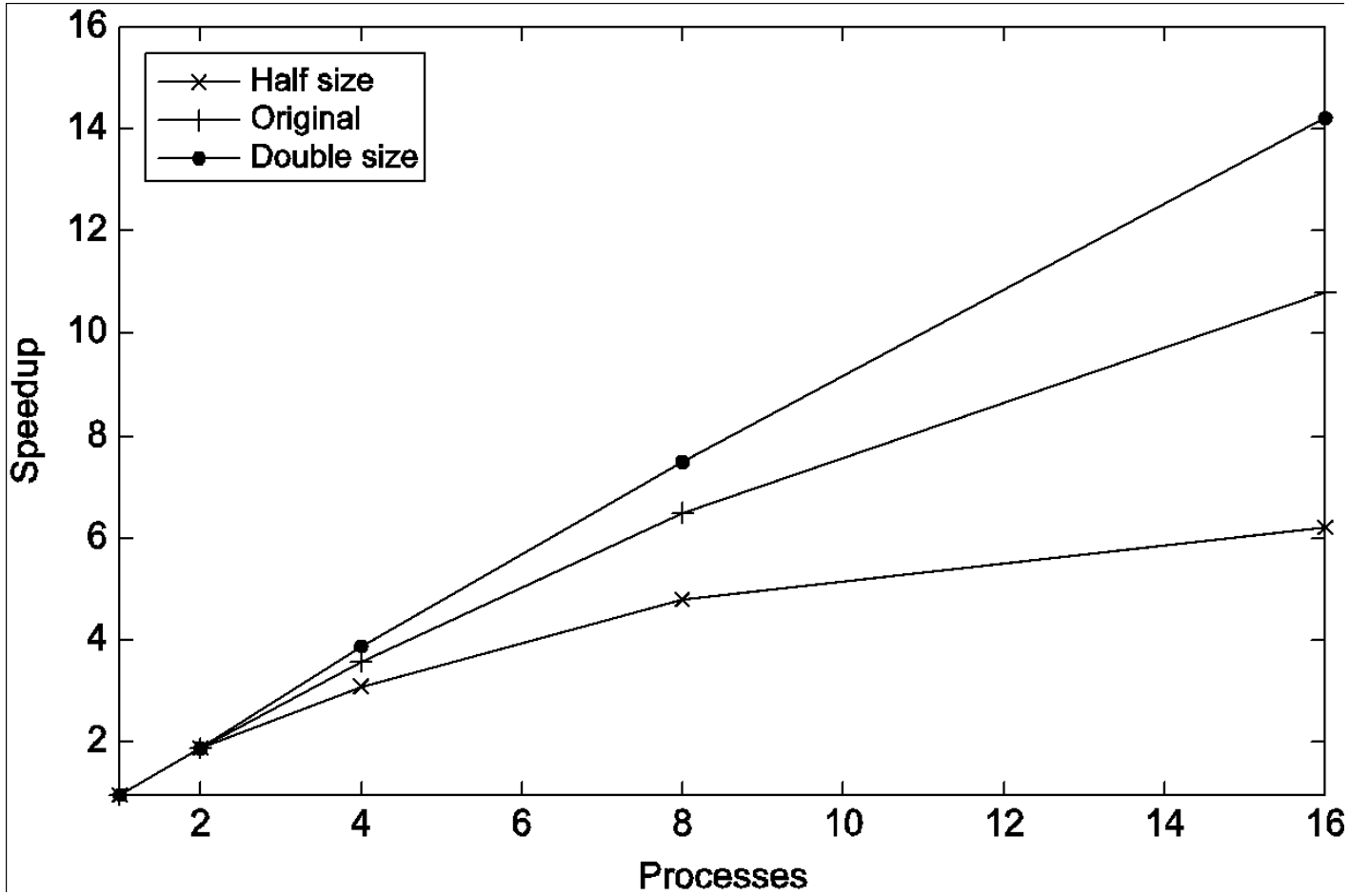
What expectations do we have?

- Ideally, when running with p processes, the program should be p times faster than when running with 1 process
- Let's define with $T_{serial}(n)$ the time of our sequential application on a problem of size n (e.g., n is the dimension of the matrix)
- Let's define with $T_{parallel}(n, p)$ the time of our parallel application on a problem of size n, when running with p processes
- Let's define with $S(n, p)$ the **speedup** of our parallel application. I.e.,

$$S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}$$

- Thus, ideally, we would like to have $S(n, p) = p$. In this case, we say our program has a **linear speedup**
- **ATTENTION:** They must be taken on the same type of cores/system. I.e., do not comput the serial time on a CPU core and the parallel time on the GPU cores
- **ATTENTION:** Let's say you implemented a new insertion sort algorithm. Shall T_{serial} be the fastest insertion sort code or the fastest sorting code in general (e.g., quicksort)?

What expectations do we have?



In general, we expect the speedup to get better when increasing the problem size n

Speedups of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Note: $T_{serial}(n)$!= $T_{parallel}(n, 1)$

- $T_{serial}(n)$ is the time of our sequential application on a problem of size n (e.g., n is the dimension of the matrix)
- $T_{parallel}(n, 1)$ is the time of our parallel application on a problem of size n, when running with 1 process
- These two implementations might be different. In general, $T_{parallel}(n, 1) >= T_{serial}(n)$

Speedup

$$S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}$$

Scalability

$$S(n, p) = \frac{T_{\text{parallel}}(n, 1)}{T_{\text{parallel}}(n, p)}$$

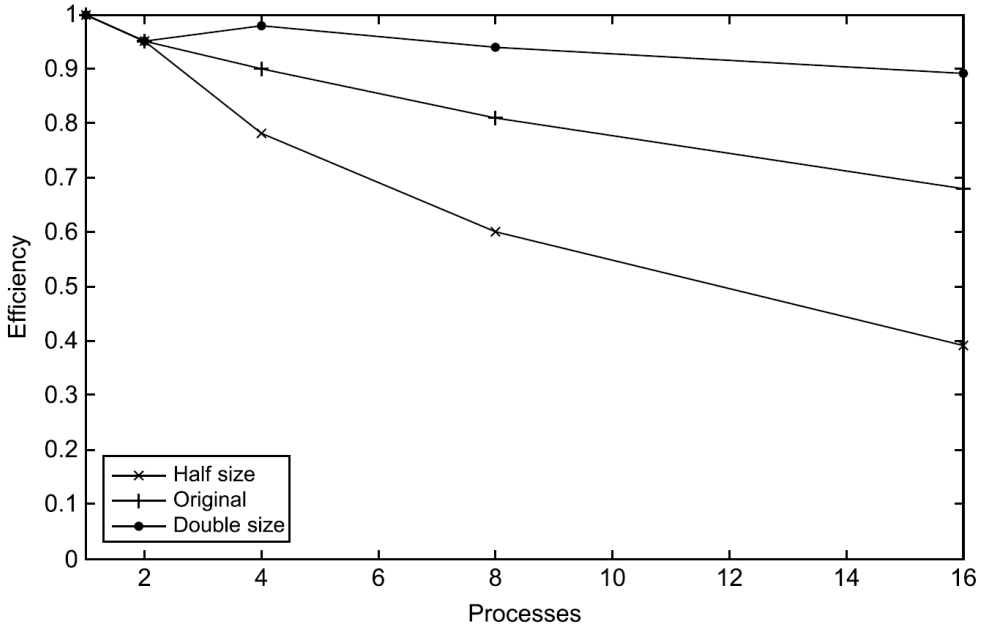
Questions?

What expectations do we have?

- **Another definition: Efficiency**

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n,p)}$$

- Ideally, we would like to have E(n,p) = 1. In practice, it is <= 1, and it gets worst with smaller problem sizes



Efficiencies of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Strong vs. Weak Scaling

Strong scaling:

- Fix the problem size, and increase the number of processes.
- If we can keep a high efficiency, our program is strong scalable

Weak scaling:

- Increase the problem size at the same rate at which you increase the number of processes.
- E.g., everytime you increase the number of processes by 2x, increase also the problem size by 2x
- If we can keep a high efficiency, our program is weak scalable

Strong vs. Weak Scaling (From Speedup Data)

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Not strongly scalable

Strong vs. Weak Scaling (From Speedup Data)

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Weakly scalable

Strong vs. Weak Scaling (From Efficiency Data)

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Not strongly scalable

Strong vs. Weak Scaling (From Efficiency Data)

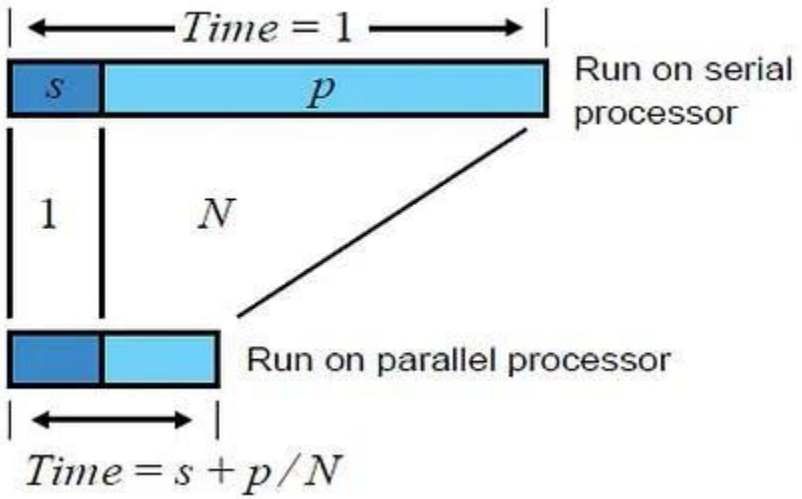
comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Weakly scalable

Questions?

Can we extrapolate expectations? Amdahl's Law

- **Intuition:** Each program has some part of it which cannot be parallelized (**serial fraction** α)
- E.g., reading/writing a file from disk, sending/receiving data over the network, serialization due to lock/unlock, etc…
- Amdahl's law says that the speedup is limited by the serial fraction.



Can we extrapolate expectations? Amdahl's Law

- Amdahl's law says that the speedup is limited by the serial fraction. I.e.,:

$$T_{parallel}(p) = (1-\alpha)T_{serial} + \alpha\frac{T_{serial}}{p}$$

- A fraction 0 ≤ α ≤ 1 can be parallelized. The remaining 1 - α has to be done sequentially.
- e.g., if α = 0, the code can't be parallelized and $T_{parallel}(p) = T_{serial}$

  if α = 1, the entire code can be parallelized and $T_{parallel}(p) = \frac{T_{serial}}{p}$(ideal speedup)

Can we extrapolate expectations? Amdahl's Law

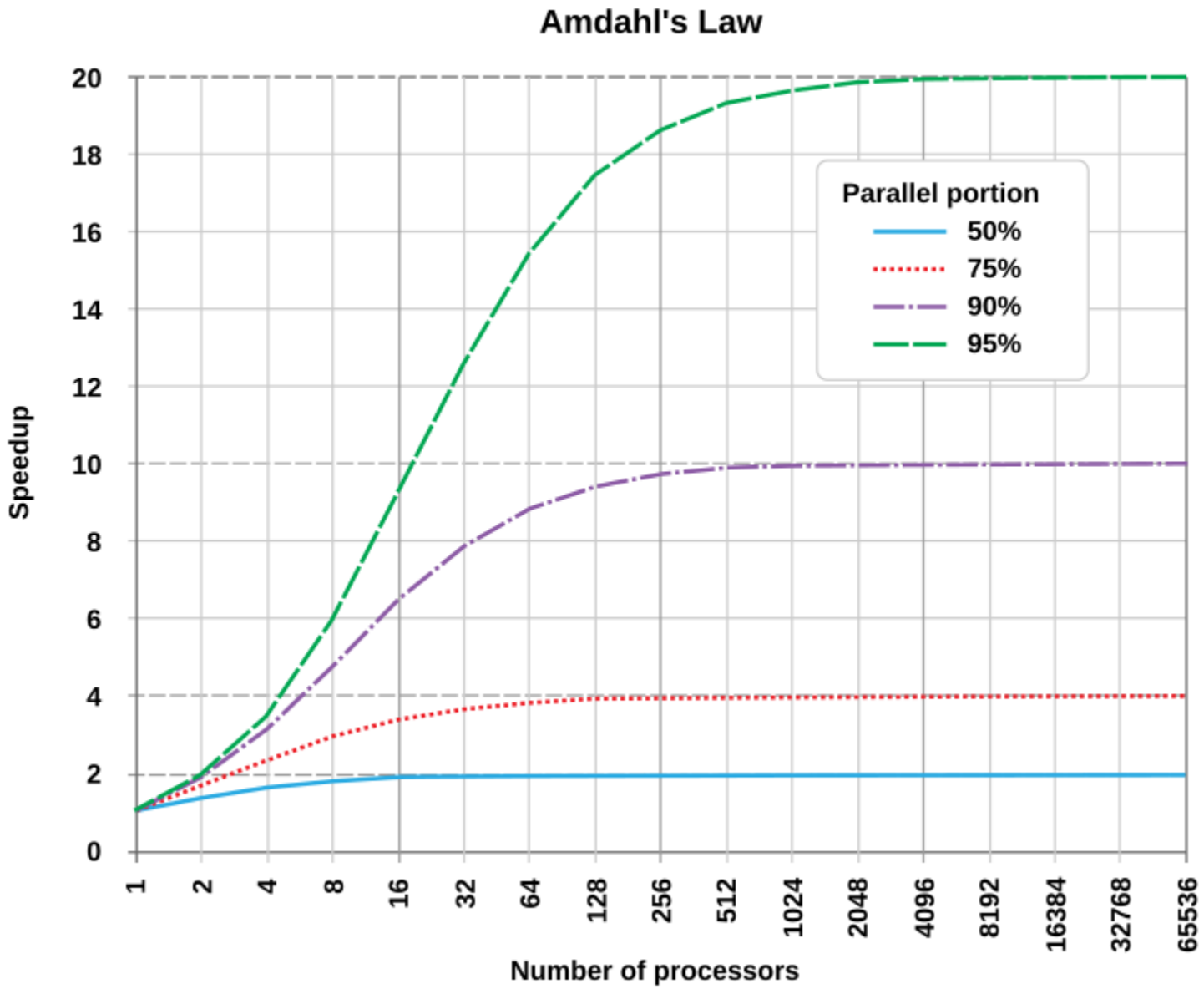
$$S(p) = \frac{T_{serial}}{(1-\alpha)T_{serial} + \alpha\frac{T_{serial}}{p}}$$

$$\lim_{p\to\infty} S(p) = \frac{1}{1-\alpha}$$

I.e.:

- if 60% of the application can be parallelized, α = 0.6, which means we can expect a speedup of at most 2.5
- if 80% of the application can be parallelized, α = 0.8, which means we can expect a speedup of at most 5
- To be able to scale up to 100000 processes, we need to have α >= 0.99999

Can we extrapolate expectations? Amdahl's Law



Questions?

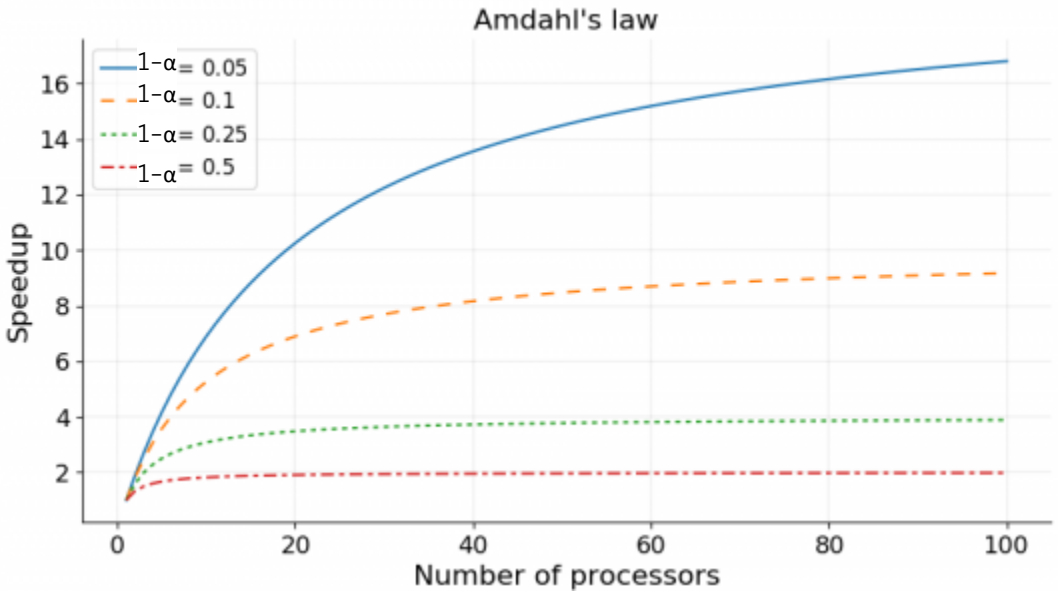
Gustafson's Law

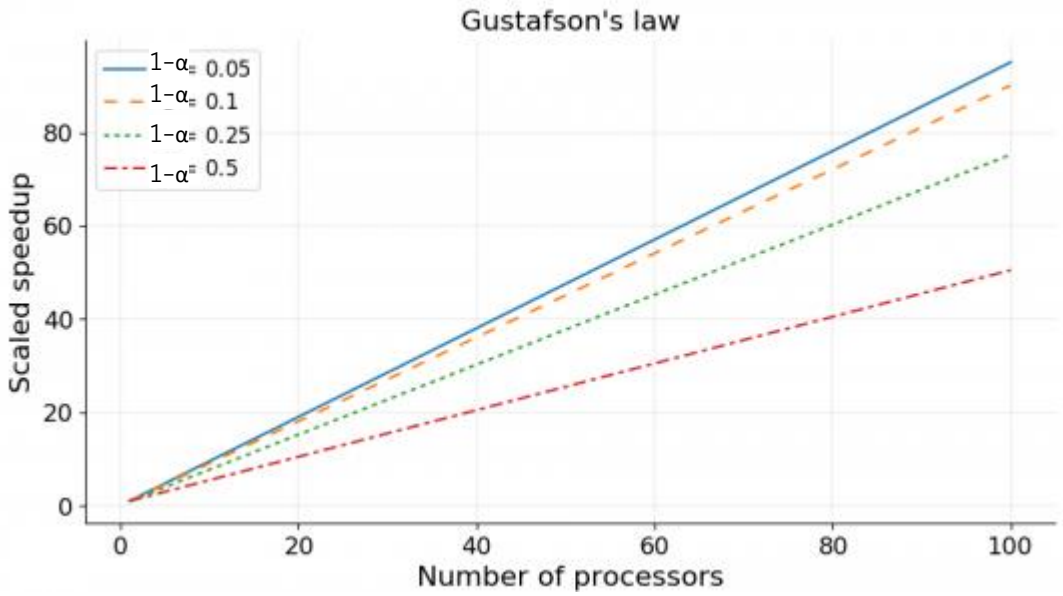
- If consider weak scaling, the parallel fraction increases with the problem size (i.e., the serial **time** remains constant, but the parallel **time** increases)
- It is also known as **scaled speedup**

$$S(n,p) = (1-\alpha) + \alpha p$$

Source: https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

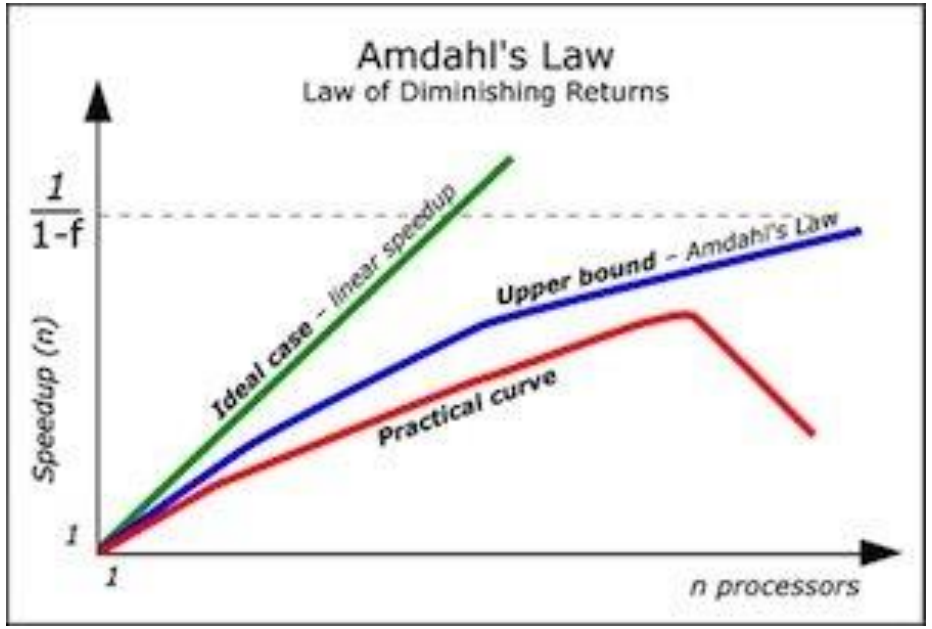
Amdahl's Law vs. Gustafson's Law





Amdahl's Law Limitations

The serial fraction could get bigger when increasing the number of processors (i.e., the runtime might increase when increasing the number of processors)



Solution: Universal Scalability Law:
https://wso2.com/blog/research/scalability-modeling-using-universal-scalability-law/

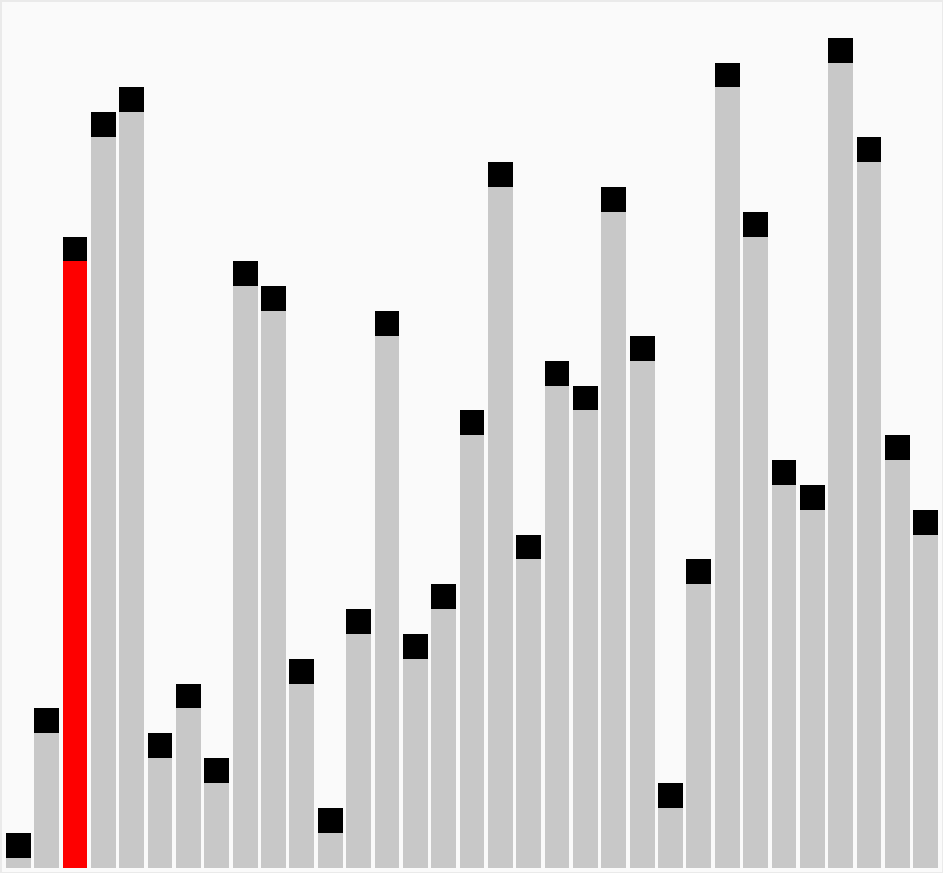
A Parallel Sorting Algorithm

Sorting

- n keys and p = comm_sz processes.
- n/p keys assigned to each process.
- No restrictions on which keys are assigned to which processes.
- When the algorithm terminates:
  - The keys assigned to each process should be sorted in (say) increasing order.
  - If $0 \le q < r < p$, then each key assigned to process q should be less than or equal to every key assigned to process r.
  - E.g.:

Process			
0	1	2	3
1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

Serial bubble sort



Serial bubble sort

```
void Bubble_sort(
        int  a[]  /* in/out */,
        int  n    /* in     */) {
    int list_length, i, temp;

    for (list_length = n; list_length >= 2; list_length--)
        for (i = 0; i < list_length-1; i++)
            if (a[i] > a[i+1]) {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
            }

}  /* Bubble_sort */
```

Inerently sequential, not many opportunities for parallelization

Odd-even transposition sort

- A sequence of phases.
- Even phases, compare swaps:

$$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots$$

- Odd phases, compare swaps:

$$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots.$$

Example

Start: 5, 9, 4, 3

Even phase: compare-swap (5,9) and (4,3)
getting the list 5, 9, 3, 4

Odd phase: compare-swap (9,3)
getting the list 5, 3, 9, 4

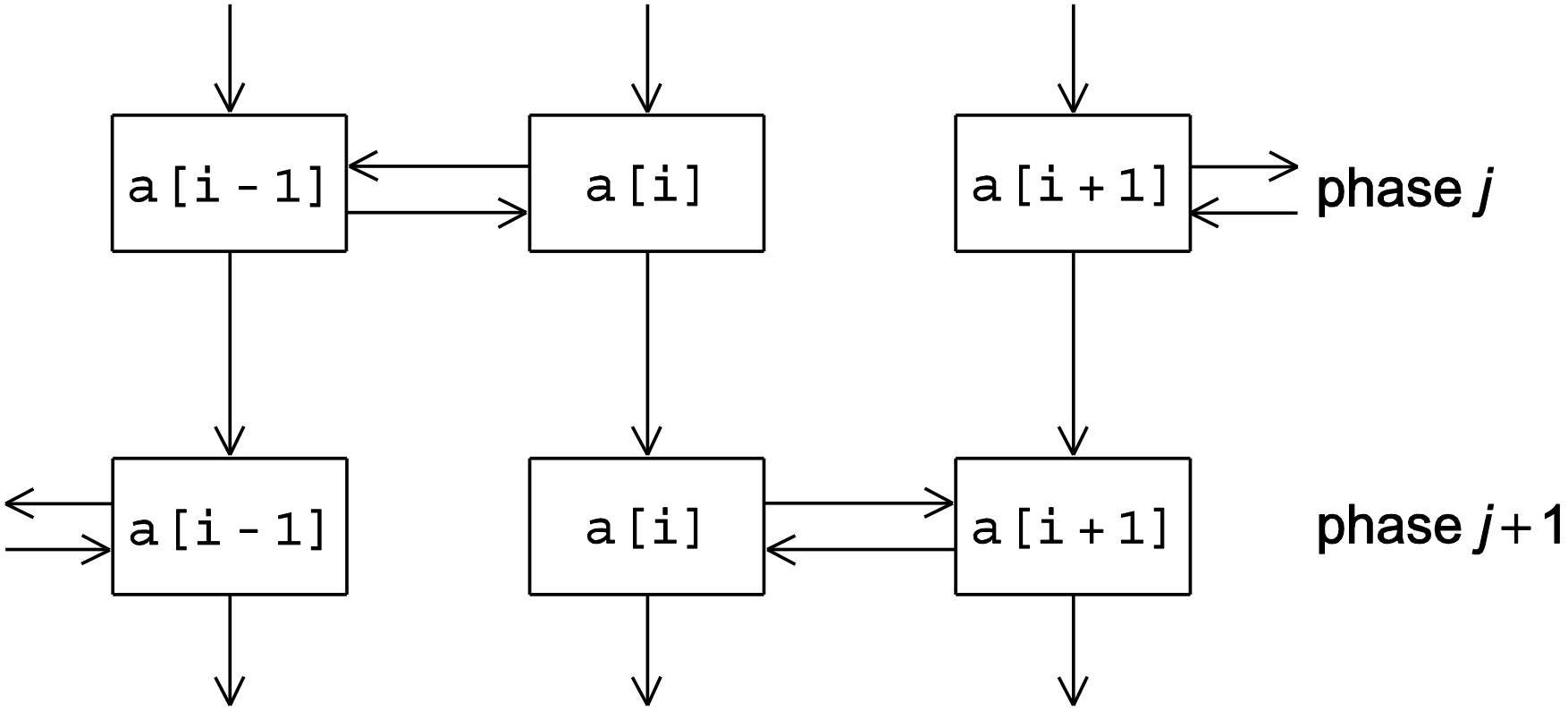
Even phase: compare-swap (5,3) and (9,4)
getting the list 3, 5, 4, 9

Odd phase: compare-swap (5,4)
getting the list 3, 4, 5, 9

Serial odd-even transposition sort

```
void Odd_even_sort(
        int  a[]  /* in/out */,
        int  n    /* in     */) {
    int phase, i, temp;

    for (phase = 0; phase < n; phase++)
        if (phase % 2 == 0) { /* Even phase */
            for (i = 1; i < n; i += 2)
                if (a[i-1] > a[i]) {
                    temp = a[i];
                    a[i] = a[i-1];
                    a[i-1] = temp;
                }
        } else { /* Odd phase */
            for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                }
        }
}  /* Odd_even_sort */
```

Communications among tasks in odd-even sort



Tasks determining a[i] are labeled with a[i].

Pseudo-code

- If number of elements to sort is equal to number of processes, each one has an element and communicates with the left/right neighbor depending on the phase being odd/even
- If n >> p:

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

Parallel odd-even transposition sort

Time	Process			
	0	1	2	3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

Compute_partner

```
if (phase % 2 == 0)          /* Even phase */
   if (my_rank % 2 != 0)        /* Odd rank */
      partner = my_rank - 1;
   else                         /* Even rank */
      partner = my_rank + 1;
else                         /* Odd phase */
   if (my_rank % 2 != 0)        /* Odd rank */
      partner = my_rank + 1;
   else                         /* Even rank */
      partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
   partner = MPI_PROC_NULL;
```

Pseudo-code

- If number of elements to sort is equal to number of processes, each one has an element and communicates with the left/right neighbor depending on the phase being odd/even
- If n >> p:

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

Parallel odd-even transposition sort

```
void Merge_low(
      int  my_keys[],      /* in/out     */
      int  recv_keys[],    /* in         */
      int  temp_keys[],    /* scratch    */
      int  local_n         /* = n/p, in */) {
   int m_i, r_i, t_i;

   m_i = r_i = t_i = 0;
   while (t_i < local_n) {
      if (my_keys[m_i] <= recv_keys[r_i]) {
         temp_keys[t_i] = my_keys[m_i];
         t_i++; m_i++;
      } else {
         temp_keys[t_i] = recv_keys[r_i];
         t_i++; r_i++;
      }
   }

   for (m_i = 0; m_i < local_n; m_i++)
      my_keys[m_i] = temp_keys[m_i];
}  /* Merge_low */
```

Questions?

Be careful with send/recv order

```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz−1) % comm_sz,
      0, comm, MPI_STATUS_IGNORE.
```

```
if (my_rank % 2 == 0) {
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz−1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE.
} else {
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz−1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE.
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
}
```

MPI_Sendrecv

- An alternative to scheduling the communications ourselves (and to using Isend/Irecv/Wait).
- Carries out a blocking send and a receive in a single call.
- The dest and the source can be the same or different.
- Especially useful because MPI schedules the communications so that the program won't hang or crash.

MPI_Sendrecv

```
int MPI_Sendrecv(
      void*          send_buf_p      /* in  */,
      int            send_buf_size   /* in  */,
      MPI_Datatype   send_buf_type   /* in  */,
      int            dest            /* in  */,
      int            send_tag        /* in  */,
      void*          recv_buf_p      /* out */,
      int            recv_buf_size   /* in  */,
      MPI_Datatype   recv_buf_type   /* in  */,
      int            source          /* in  */,
      int            recv_tag        /* in  */,
      MPI_Comm       communicator    /* in  */,
      MPI_Status*    status_p        /* in  */);
```