Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV

Design and Implementation of Parallel Applications

Summer Term 2023 | 05.06.2023

Barry Linnert

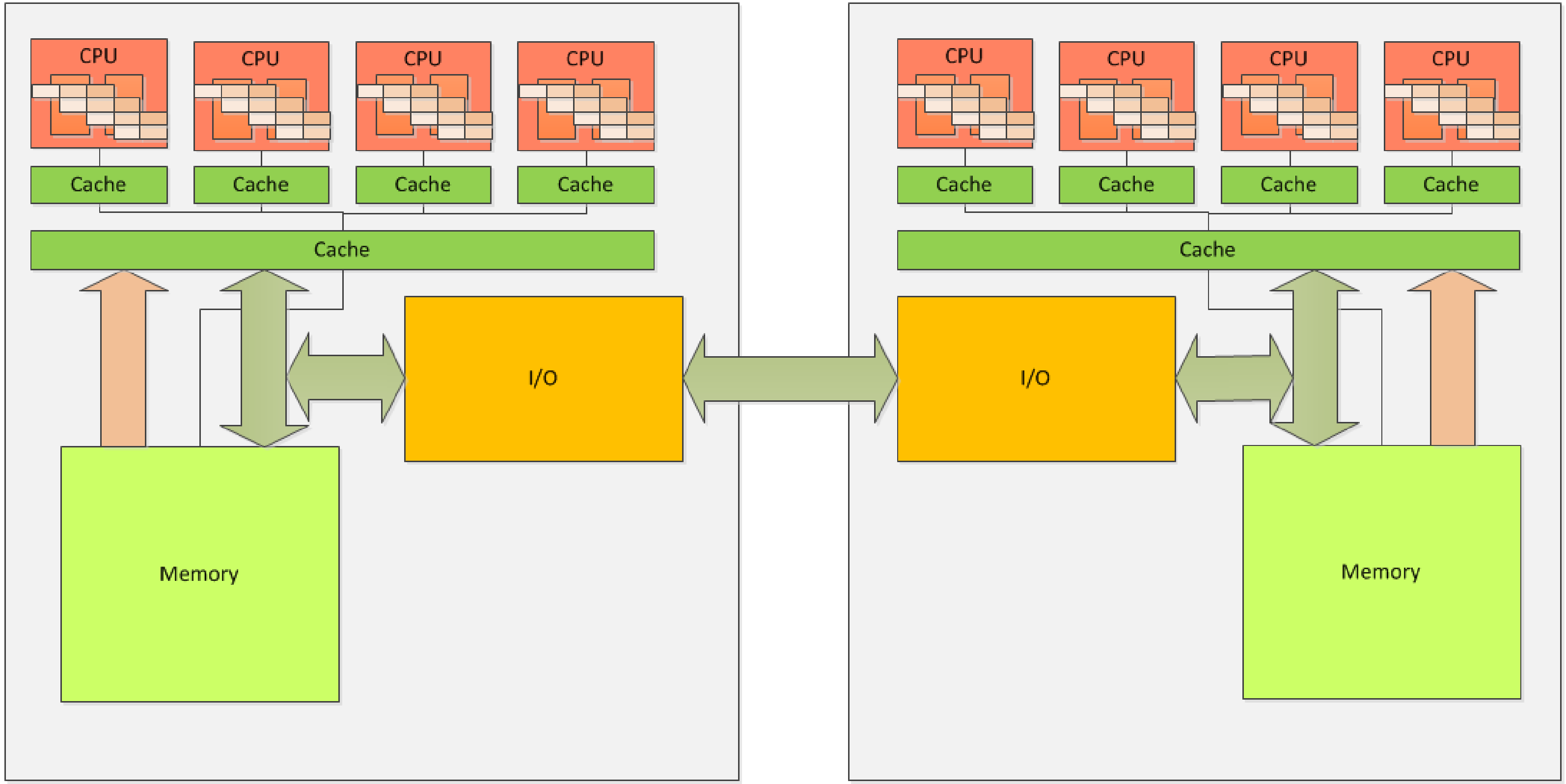
Objectives of Today‘s Lecture

- Matrix multiplication
- Sieve of Eratosthenes

Concepts of Non-sequential and Distributed Programming

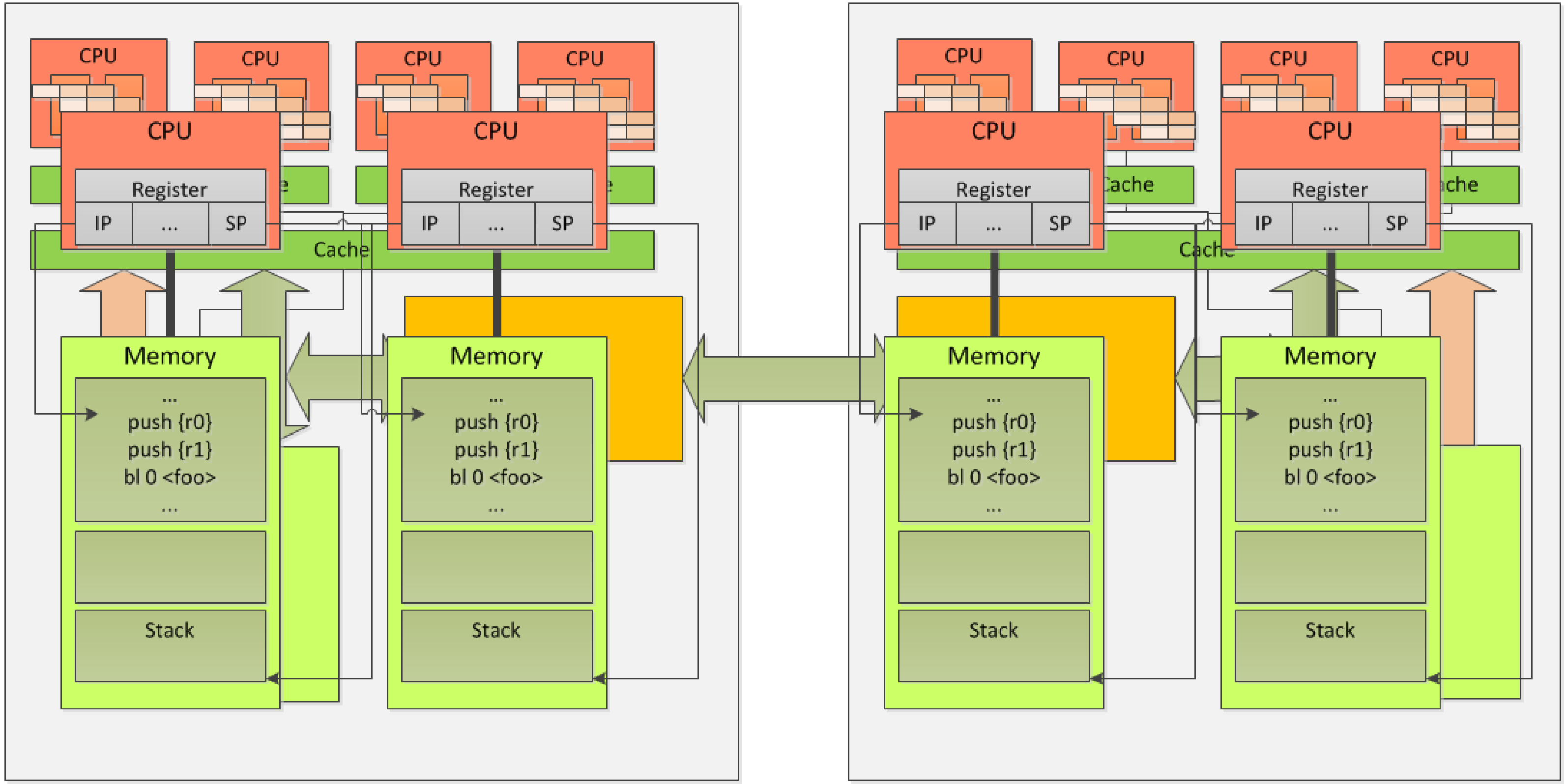
DESIGN AND IMPLEMENTATION OF PARALLEL APPLICATIONS

Machine Model



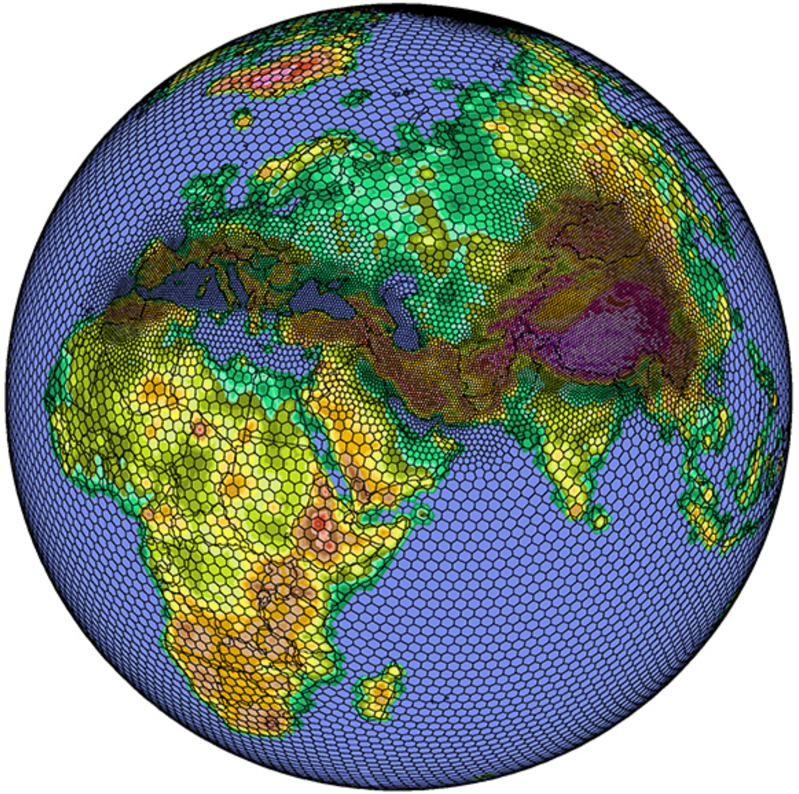
Freie Universität Berlin

Machine and Execution Model



MPI and MPI-2

- With MPI and MPI-2 the foundation to design and implement parallel programs using message passing is given.
- Thus, the design methodology by Ian Foster is applicable.
- The functions coming with the MPI-2 standard extension can be used additionally to design parallel programs with dynamic runtime behavior, such as:
  - Ocean Land Atmosphere Model
    - http://olam-soil.org



Evaluation of Parallel Applications

- Let be

  $T(1)$ the execution time on one processor

  $T(p)$ the execution time on a p processor system

- The gain by parallel computing is expressed by

  $S(p) := T(1) / T(p)$ *Speed-up*

- Normalizing the Speed-up by dividing by the number *p* of processors is defined as the *efficiency*:

  $E(p) := S(p) / p$ *Efficiency*

Amdahl's Law

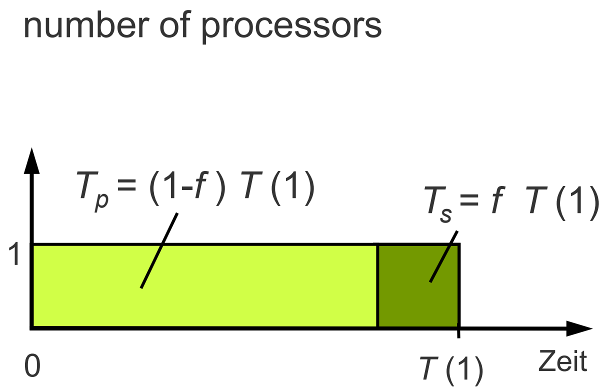
- Parallel programs also contain sequential parts.
- Splitting the execution time into a sequential and a parallelizable part yields:

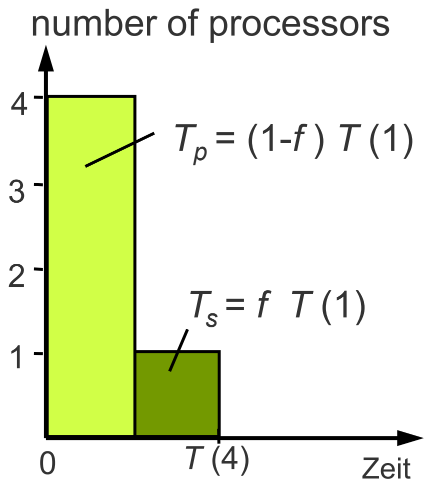
  $T(1) = T_s + T_p$

- Let $f := T_s / (T_s + T_p)$, $(0 \leq f \leq 1)$ be the sequential fraction of the program. Then we get for the execution time:

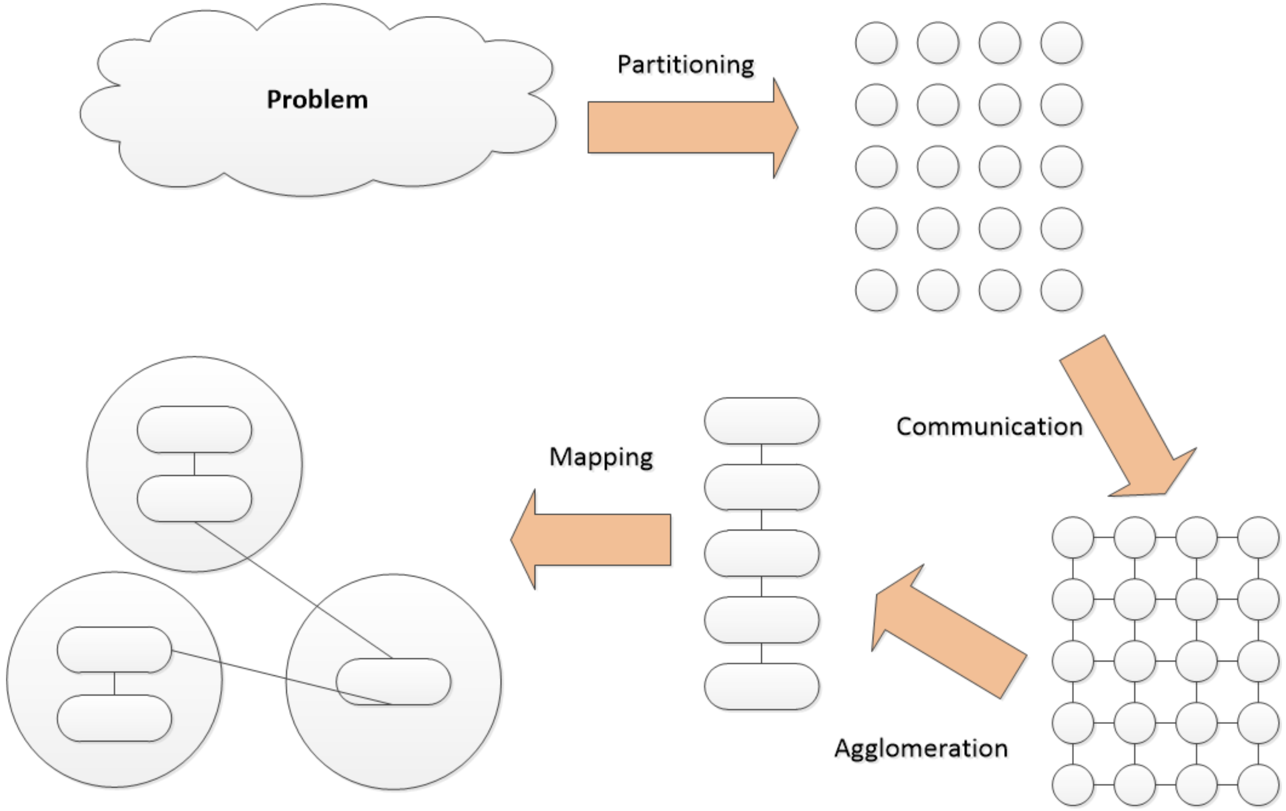
$$T(p) = fT(1) + \frac{(1-f)T(1)}{p} = T_s + \frac{T_p}{p} \quad \textit{(Amdahl's law)}$$

Amdahl's Law II





Foster's Design Methodology



Task/Channel (Programming) Model

- The Task/Channel (Programming) Model serves as foundation for Forster‘s design methodology.
- A Task of the task/channel model is a part of the application with its own address space (process).
- Tasks can exchange data via messages using channels.
- A channel is a message queue connecting two specific tasks.
- If a task wants to receive a message, the task waits until the message is received (is blocked).
- Messages are sent immediately by the sender. The sender does not wait until the message is received.
- Thus, the task/channel model implements synchronous receive and asynchronous sending.

Concepts of Non-sequential and Distributed Programming

EXAMPLE: MATRIX MULTIPLICATION

Matrix Multiplication

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} + A_{13} \cdot B_{32} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + A_{23} \cdot B_{31} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} + A_{23} \cdot B_{32} \end{pmatrix}$$

```c
// sequential program matrix mult.
// Rauber, Ruenger: Parallele und vert. Prg.

#include <stdio.h>


double MA[100][100], MB[100][100];
double MC[100][100];
int i, row, col, size = 100;

void read_input ()
{
  int j;

  for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
      MA[i][j] = (double)(i + j) + 1.0;

  for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
      MB[i][j] = (double)(i + j) + 1.0;
}
```

```c
void write_output ()
{
  int j;

  for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
      printf ("%f ", MC[i][j]);
    printf ("\n");
}


int main () {
  read_input (); // MA, MB

  ...

  write_output (); // MC

  return 0;
}
```

source code: 13-00.c

```c
// sequential program matrix mult.
// Rauber, Ruenger: Parallele und vert. Prg.

#include <stdio.h>


double MA[100][100], MB[100][100];
double MC[100][100];
int i, row, col, size = 100;

void read_input ()
{
  ...
}

void write_output ()
{
  ...
}
```

```c
int main () {
  read_input (); // MA, MB

  for (row = 0; row < size; row++) {
    for (col = 0; col < size; col++)
      MC[row][col] = 0.0;
  }

  for (row = 0; row < size; row++) {
    for (col = 0; col < size; col++)
      for (i = 0; i < size; i++)
        MC[row][col] += MA[row][i] *
         MB[i][col];
  }

  write_output (); // MC
  return 0;
}
```

source code: 13-00.c

Matrix Multiplication – Partitioning

- Which (preferably independent) tasks (elements of work) can be identified?

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} + A_{13} \cdot B_{32} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + A_{23} \cdot B_{31} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} + A_{23} \cdot B_{32} \end{pmatrix}$$

- The calculation of every element of the target matrix can be performed independently of all other calculations.

Matrix Multiplication – Communication

- What is the data that has to be transferred (from which task to which other task)?
- Which transfer of data depends on which other activities (calculation, communication)?

- As the root task is responsible for initialization of the data, it has to provide the data to all other tasks.
- Thus, from the root task, one row of the output matrix A and one column of the output matrix B have to be transferred to a task.
- After the calculation, the results of the different tasks must be transferred back to the root task.

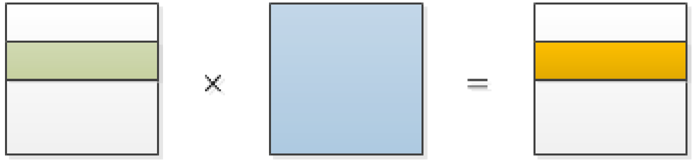
Matrix Multiplication – Agglomeration

- Which tasks can be combined as long as the number of available processors is (much) smaller than the number of combined tasks (now processes)?
- How can the aggregation of the tasks be used to reduce communication between the processes?

- For the agglomeration of the tasks the calculation part is not important, because all calculations are independent.
- There is no data dependency in calculating the results.
- When the data is distributed, the data that is relevant for the specific task may also be needed by other tasks:
  - all elements of the row of the initial matrix A
  - all elements of the column of the initial matrix B

Matrix Multiplication – Agglomeration II

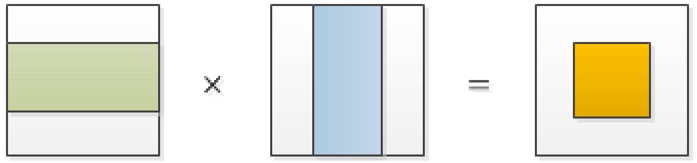
- Tasks using the same data should be combined to form a process.
- Thus, the calculations of the elements of a row (or column) of matrix A may be combined.



- All elements of the initial matrix B must be transferred for the calculations.

Matrix Multiplication – Agglomeration III

- In order to reduce the data to be transferred (especially with uneven matrixes) the parts of the matrix representing the results can be shaped to form square.
- Thus, only a part of the second matrix B is to be transferred.



Matrix Multiplication – Mapping

- Which combined tasks (processes) should be assigned to which processor?

- How can communication costs between the processes can be reduced?
  - There is only communication from the root process to the others processes and back.
  - There is no further communication between the processes.
  - Thus, the root process should be located to reduce the communication costs to all of the other processes involved. There are no further requirements coming with the communication.
- Are there any requirements coming with the distribution of (calculation) load?
  - If the agglomeration produces as many pooled tasks as there are processors available and the corresponding processes all do the same work (no sparsely populated matrices), the load is balanced.

```c
// simple parallel matrix mult
// https://stackoverflow.com/questions/
// 41575243/matrix-multiplication-using-
// mpi-scatter-and-mpi-gather

#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stddef.h>
#include "mpi.h"
#define N 4

void print_results (char *prompt,
           int a[N][N]){
  int i, j;
  printf ("\n\n%s\n", prompt);
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
      printf(" %d", a[i][j]);
    printf ("\n");
  }
  printf ("\n\n");
}
```

```c
int main(int argc, char *argv[]) {
  int i, j, k, rank, size, tag = 99;
  int blksz, sum = 0;
  int a[N][N]={...};
  int b[N][N]={...};
  int c[N][N];
  int aa[N],cc[N];

  // init MPI

  //scatter rows of first matrix

  //broadcast second matrix to all procs

  //perform vector multiplication

  // finalizing MPI
}
```

source code: 13-01.c

```c
// simple parallel matrix mult
// https://stackoverflow.com/questions/
// 41575243/matrix-multiplication-using-
// mpi-scatter-and-mpi-gather

#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stddef.h>
#include "mpi.h"
#define N 4

void print_results (char *prompt,
          int a[N][N]){
  int i, j;
  printf ("\n\n%s\n", prompt);
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
      printf(" %d", a[i][j]);
    printf ("\n");
  }
  printf ("\n\n");
}
```

```c
int main(int argc, char *argv[]){
  ...

  // init MPI
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  //scatter rows of first matrix

  //broadcast second matrix to all procs

  //perform vector multiplication

  // finalizing MPI
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
  if (rank == 0)
    print_results("C = ", c);
}
```

source code: 13-01.c

```
// simple parallel matrix mult
// https://stackoverflow.com/questions/
// 41575243/matrix-multiplication-using-
// mpi-scatter-and-mpi-gather

#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stddef.h>
#include "mpi.h"
#define N 4

void print_results (char *prompt,
         int a[N][N]){
  int i, j;
  printf ("\n\n%s\n", prompt);
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
      printf(" %d", a[i][j]);
    printf ("\n");
  }
  printf ("\n\n");
}
```

```
int main(int argc, char *argv[]) {
  ...

  // init MPI

  //scatter rows of first matrix
  MPI_Scatter(a, N*N/size, MPI_INT, aa,
     N*N/size, MPI_INT,0,MPI_COMM_WORLD);

  //broadcast second matrix to all procs
  MPI_Bcast(b, N*N, MPI_INT, 0,
        MPI_COMM_WORLD);

  MPI_Barrier(MPI_COMM_WORLD);

  //perform vector multiplication

  // finalizing MPI
}
```

source code: 13-01.c

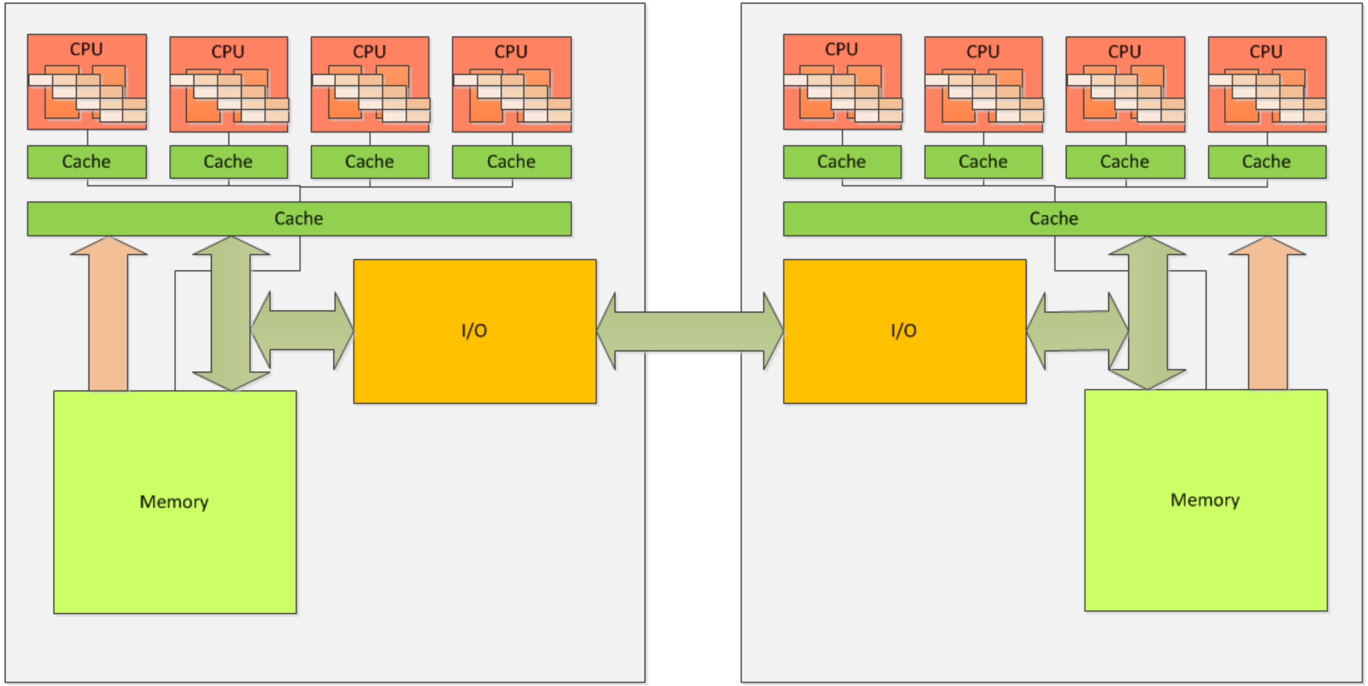
```c
// simple parallel matrix mult
// https://stackoverflow.com/questions/
// 41575243/matrix-multiplication-using-
// mpi-scatter-and-mpi-gather

#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stddef.h>
#include "mpi.h"
#define N 4

void print_results (char *prompt,
          int a[N][N]){
  int i, j;
  printf ("\n\n%s\n", prompt);
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
      printf(" %d", a[i][j]);
    printf ("\n");
  }
  printf ("\n\n");
}
```

```c
int main(int argc, char *argv[]) {
  ...

  // init MPI

  //scatter rows of first matrix

  //broadcast second matrix to all procs

  //perform vector multiplication
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
      sum = sum + aa[j] * b[j][i];
    cc[i] = sum;
    sum = 0;
  }
  MPI_Gather(cc, N*N/size, MPI_INT, c,
   N*N/size, MPI_INT, 0, MPI_COMM_WORLD);

  // finalizing MPI
}
```

source code: 13-01.c

Matrix Multiplication – Optimization

- Are there any requirements or approaches that may lead to optimizations?
- Caches!



Concepts of Non-sequential and Distributed Programming

EXAMPLE: SIEVE OF ERATOSTHENES

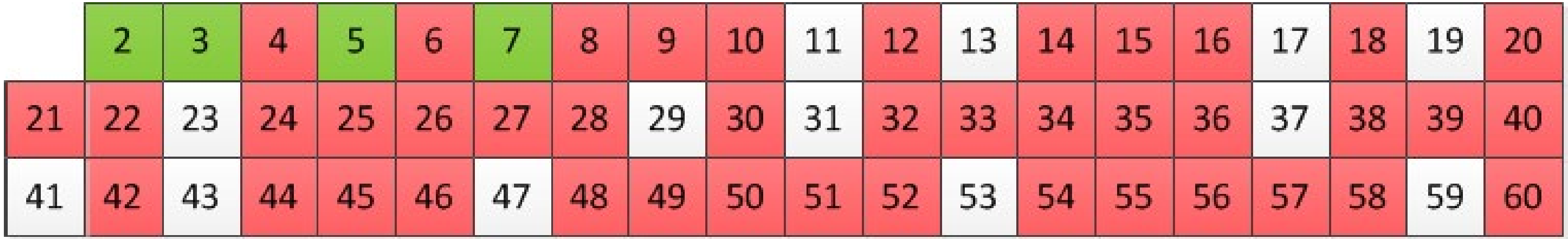
Sieve of Eratosthenes

- Eratosthenes of Cyrene (276/273 - 194 b.c.[e.])
- The sieve of Eratosthenes is an algorithm to determine all prime numbers up to a certain maximum by exclusion.

wikipedia.org

- Algorithm:
  1. Sort all numbers from 2 to the maximum *n* in a list and mark them as unmarked.
  2. Take the first (unmarked) number (2 for the first iteration) as key *k*.
  3. Repeat until $k^2 > n$
     I. Mark all multiples of *k* between k^2 and *n,*
     II. Find the smallest unmarked number greater than *k* and take it as the new key *k.*
  4. The unmarked numbers are prime numbers.

Sieve of Eratosthenes



	🟩 2	🟩 3	🟥 4	🟩 5	🟥 6	🟩 7	🟥 8	🟥 9	🟥 10	⬜ 11	🟥 12	⬜ 13	🟥 14	🟥 15	🟥 16	⬜ 17	🟥 18	⬜ 19	🟥 20
🟥 21	🟥 22	⬜ 23	🟥 24	🟥 25	🟥 26	🟥 27	🟥 28	⬜ 29	🟥 30	⬜ 31	🟥 32	🟥 33	🟥 34	🟥 35	🟥 36	⬜ 37	🟥 38	🟥 39	🟥 40
⬜ 41	🟥 42	⬜ 43	🟥 44	🟥 45	🟥 46	⬜ 47	🟥 48	🟥 49	🟥 50	🟥 51	🟥 52	⬜ 53	🟥 54	🟥 55	🟥 56	🟥 57	🟥 58	⬜ 59	🟥 60

Sieve of Eratosthenes – Partitioning

- Which (preferably independent) tasks (elements of work) can be identified?

- The algorithm invites a loop with some complex operations of processing.
- Thus, start with the operations to process on the key k.
- This is called function-based decomposition:
  - Determination of the multiple of the key k.
  - Scrolling through the list.
  - Marking the corresponding number.

- Is this partitioning reasonable? What about the cost-benefit ratio?
  - Coordination of all tasks after each step for all multiples of all keys is needed.
  - That comes with significant overhead.

Sieve of Eratosthenes – Partitioning II

- The other approach would be a domain- or data-based decomposition:
  - Processing all operations of the loop for a given value of the key k.

- Cost-benefit ratio?
  - Transfer of the multiple of k after marking is needed or
  - at the least after marking all of the multiples of the key k.
  - In order to prevent processing marked keys, the processes have to be synchronized after the processing of a key k.

- As the domain-based decomposition approach leads to a higher amount of tasks this approach is used for further consideration.

Sieve of Eratosthenes – Communication

- What is the data that has to be transferred (from which task to which other task)?
- Which transfer of data depends on which other activities (calculation, communication)?

- From the root task the list of unmarked numbers must be transferred to all tasks.
- Transfer of the value of the multiples of the key k from all to all other tasks.
- Gathering of the unmarked numbers from all tasks to the root task.

Sieve of Eratosthenes – Agglomeration

- Which tasks can be combined as long as the number of available processors is (much) smaller than the number of combined tasks (now processes)?
- How can the aggregation of the tasks be used to reduce communication between the processes?

- Combination of all calculations for a key k.
  - Reduce the communication as all of the multiples of the key k are transferred.
  - BUT: The determination of new keys k depend on the result of marking the multiples of a previous handled key. Thus, receiving a value too late may lead to additional handling of an otherwise marked multiple of a key k.

Sieve of Eratosthenes – Agglomeration II

- Combination of block of tasks processing different keys (block data decomposition).

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	process 0
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	process 1
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	process 2

- Reduces the communication costs heavily, at the cost of redundant calculation.

Sieve of Eratosthenes – Mapping

- Which combined tasks (processes) should be assigned to which processor?
- How can communication costs between the processes can be reduced?
  - There is communication from the root process to the others processes and back to distribute the data and gather the results.
  - Thus, the root process should be located to reduce the communication costs to all of the other processes involved.
  - Depending on the approach and level of agglomeration there is communication from all to all processes transferring the intermediate results. This communication is not to be influenced by the mapping decision.
- Are there any requirements coming with the distribution of (calculation) load?
  - If the agglomeration produces as many combined tasks as there are processors available and the corresponding processes all do the same work (effort of marking the multiples), the load is balanced.

```
// Sieve of Erastosthenes
// Quinn: Par. Prg. in C with MPI + OMP
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include "MyMPI.h"

#define MIN (a, b) ((a)<(b) ? (a) : (b))

int main (int argc, char *argv[]){
  int count, first, global_count,
    high_value, i, id, index,
    low_value, n, p, proc0_size, prime,
    size;
  double elapsed_time;
  char *marked;
  ...
  // Start the timer
  ...
  // Fig. out proc's share of the array
  ...
  // Bail out if all the primes used for
  // sieving are not all held by proc 0
```

```
// Allocate this
// proc's share
// of array

// do the work

// counting results

// Stop timer

// Print results

MPI_Finalize ();
return 0;}
```

source code: 13-02.c

```c
// Sieve of Erastosthenes
// Quinn: Par. Prg. in C with MPI + OMP
...

...

int main (int argc, char *argv[]){
    ...

    MPI_Init (&argc, &argv);
    // Start the timer
    MPI_Barrier (MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime ();

    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);

    // Fig. out proc's share of the array
    ...
    // Bail out if all the primes used for
    // sieving are not all held by proc 0
    ...
```

```c
// Allocate this
// proc's share
// of array

// do the work

// counting results

// Stop timer

// Print results

MPI_Finalize ();
return 0;}
```

source code: 13-02.c

```
// Sieve of Erastosthenes
// Quinn: Par. Prg. in C with MPI + OMP
...

...

int main (int argc, char *argv[]){
   ...

   // Start the timer
   ...
   n = atoi (argv[1]);

   // Fig. out proc's share of the array
   low_value = 2 + BLOCK_LOW (id, p, n-1);
   high_value = 2 + BLOCK_HIGH (id, p,
             n-1);
   size = BLOCK_SIZE (id, p, n-1);

   // Bail out if all the primes used for
   // sieving are not all held by proc 0
   ...
```

```
// Allocate this
// proc's share
// of array

// do the work

// counting results

// Stop timer

// Print results

MPI_Finalize ();
return 0;}
```

source code: 13-02.c

```
// Sieve of Erastosthenes
// Quinn: Par. Prg. in C with MPI + OMP
...

...

int main (int argc, char *argv[]){
   ...

   // Start the timer
   ...
   // Fig. out proc's share of the array
   ...
   // Bail out if all the primes used for
   // sieving are not all held by proc 0
   proc0_size = (n-1)/p;
   if ((2 + proc0_size) <
            (int) sqrt ((double) n)) {
      if (!id) printf ("Too many processes
            \n");
      MPI_Finalize ();
      exit (1);
   }
```

```
// Allocate this
// proc's share
// of array

// do the work

// counting results

// Stop timer

// Print results

MPI_Finalize ();
return 0;}
```

source code: 13-02.c

```
// Sieve of Erastosthenes
// Quinn: Par. Prg. in C with MPI + OMP
...

...

int main (int argc, char *argv[]){
   ...

   // Start the timer
   ...
   // Fig. out proc's share of the array
   ...
   // Bail out if all the primes used for
   // sieving are not all held by proc 0
   ...
```

```
   // Allocate this
   // proc's share
   // of array
   marked = (char *) malloc (size);

   if (marked == NULL) {
     printf ("Cannot alloc memory \n");
     MPI_Finalize ();
     exit (1);
   }

   for (i = 0; i < size; i++)
     marked[i] = 0;
   if (!id) index = 0;
   prime = 2;

   // do the work
   // counting results
   // Stop timer
   // Print results
   MPI_Finalize ();
   return 0;
}
```

source code: 13-02.c

```c
// Sieve of Erastosthenes
// Quinn: Par. Prg. in C with MPI + OMP
...

...

int main (int argc, char *argv[]){
   ...

   // Start the timer
   ...
   // Fig. out proc's share of the array
   ...
   // Bail out if all the primes used for
   // sieving are not all held by proc 0
   ...
```

```c
...
// do the work
do {
   if (prime * prime > low_value)
      first = prime * prime - low_value;
   else {
      if (!(low_value % prime)) first=0;
      else first = prime -
         (low_value % prime);
   }

   for (i = first; i < size; i += prime)
      marked[i] = 1;

   // proc 0 gets the next prime for all
   if (!id) {
      while (marked[++index]);
      prime = index + 2;
   }
   MPI_Bcast (&prime, 1, MPI_INT, 0,
         MPI_COMM_WORLD);
} while (prime * prime <= n);
...
```

source code: 13-02.c

```
// Sieve of Erastosthenes
// Quinn: Par. Prg. in C with MPI + OMP
...

...

int main (int argc, char *argv[]){
  ...

  // Start the timer
  ...
  // Fig. out proc's share of the array
  ...
  // Bail out if all the primes used for
  // sieving are not all held by proc 0
  ...
```

```
  // Allocate this proc's share of array
  // do the work
  // counting results
  count = 0;
  for (i = 0; i < size; i++)
    if (!marked[i]) count++;
  MPI_Reduce (&count, &global_count, 1,
          MPI_INT, MPI_SUM, 0,
          MPI_COMM_WORLD);

  // Stop timer
  elapsed_time += MPI_Wtime ();

  // Print results
  if (!id) {
    printf ("%d primes are less than or
       equal to %d \n", global_count, n);
    printf ("Total elapsed time: %10.6f
         \n", elapsed_time);
  }
  MPI_Finalize ();
  return 0;
}
```

source code: 13-02.c

Sieve of Eratosthenes – Optimization

- In order to reduce the redundant work, the tasks can be combined is a round-robin-like approach. So, the first keys are distributed over all of the processes available and the chance the multiples are determined before the next amount of keys are distributed is increased.

Concepts of Non-sequential and Distributed Programming

NEXT LECTURE

Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Design and Implementation of Parallel Applications II

APL IV: Concepts of Non-sequential and Distributed Programming (Summer Term 2023)