Lecture 11

Let’s write an outline main function with stubs to reflect this algorithm

Keep things simple with dummy type definition

```
typedef int LOCAL_LIST_T;

typedef int KEY_T;
```

```
/* sort_1.c -- level 1 version of sort program
 *
 * Input: none
 * Output:  messages indicating flow of control through program
 *
 * See Chap 10, pp. 226 & ff in PPMPI.
 */
#include <stdio.h>
#include <string.h>
#include "mpi.h"
#include "cio.h"
#include "sort_1.h"

int       p;
int       my_rank;
MPI_Comm  io_comm;

main(int argc, char* argv[]) {
    LOCAL_LIST_T  local_keys;
    int           list_size;
    int           error;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);
    Cache_io_rank(MPI_COMM_WORLD, io_comm);
```

```
list_size = Get_list_size();

/* Return negative if Allocate failed */
error = Allocate_list(list_size, &local_keys);

Get_local_keys(&local_keys);
Print_list(&local_keys);
Redistribute_keys(&local_keys);
Local_sort(&local_keys);
Print_list(&local_keys);
    MPI_Finalize();}
 /* main */

int Get_list_size() {
    Cprintf(io_comm,"","%s", "In Get_list_size");
    return 0;
} /* Get_list_size */
```

```
/* Return value negative indicates failure */
int Allocate_list(int list_size,
    LOCAL_LIST_T* local_keys) {

    Cprintf(io_comm,"","%s", "In Allocate_key_list");
    return 0;
} /* Allocate_list */

void Get_local_keys(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm,"","%s", "In Get_local_keys");
} /* Get_local_keys */

void Redistribute_keys(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm,"","%s", "In Redistribute_keys");
} /* Redistribute_keys */

void Local_sort(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm,"","%s","In Local_sort");
} /* Local_sort */

void Print_list(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm,"","%s","In Print_list");
} /* Print_list */
```

```
/* sort_1.h -- header file for sort_1.c */
#ifndef SORT_H
#define SORT_H

#define KEY_MIN 0
#define KEY_MAX 32767
#define KEY_MOD 32768

typedef int KEY_T;
typedef int LOCAL_LIST_T;

#define List_size(list) (0)
#define List_allocated_size(list) (0)

int Get_list_size(void);
int Allocate_list(int list_size,
    LOCAL_LIST_T* local_keys);
void Get_local_keys(LOCAL_LIST_T* local_keys);
void Redistribute_keys(LOCAL_LIST_T* local_keys);
void Local_sort(LOCAL_LIST_T* local_keys) ;
void Print_list(LOCAL_LIST_T* local_keys) ;
#endif
```

Test stage 1 programs

Even though it’s very simple, testing it will reveal typos, simple mistakes etc.

After this is done, the rest is just a matter of filling in the subprograms.

Since input is a major problem, first write a version with hardwired input.

In this initial version, also restrict yourself to small list size, since you will want to print the list often as you write and debug your program. For example, consider only lists of length 5*p, where p is number of processes

```
/*********************************************************************/
int Get_list_size(void) {
    Cprintf(io_comm,"","%s","In Get_list_size");
    return 5*p;
} /* Get_list_size */
```

We can’t actually allocate the list yet, as we have not decided on the actual structure.

We do want members of the structure to record number of keys and space allocated. So, we provisionally modify LOCAL_LIST_T

In sort.h, we now have:

```
typedef struct {
    int allocated_size;
    int local_list_size;
    int keys;  /* dummy member */
} LOCAL_LIST_T;

/* Assume list is a pointer to a struct of type
 * LOCAL_LIST_T */
#define List_size(list) ((list)->local_list_size)
#define List_allocated_size(list) ((list)->allocated_size)
```

We will use a random number generator to generate the keys on each process. For now define empty function Insert_key

```
/*****************************************************************/
void Get_local_keys(LOCAL_LIST_T* local_keys) {
   int i;

   /* Seed the generator */
   srand(my_rank);

   for (i = 0; i < List_size(local_keys);  i++)
      Insert_key(rand() % KEY_MOD, i, local_keys);
} /* Get_local_keys */

/*****************************************************************/
void Insert_key(KEY_T key, int i,
   LOCAL_LIST_T* local_keys) {
} /* Insert_key */
```

Stage 2 complete. Very minor changes from stage 1.

Now is the time to decide on how to send the original local keys from each process to the appropriate address

This exchange of data will be an example of an:

all-to-all scatter/gather or total exchange

MPI provides dedicated functions for this, which we will use

MPI_Alltoall and MPI_Alltoallv

Say each process determines how many elements it must send to the other process

This information must then be exchanged among processes so that we can then follow with the actual sending of keys

After keys are locally sorted, each process knows how many have to be send to the other processes. But that information must be passed to the recipients. For example, information in shaded column must go to process 2 etc.

Source process	# of Keys for 0	# of Keys for 1	# of Keys for 2	# of Keys for 3
0	5	7	6	7
1	4	6	8	7
2	7	4	9	3
3	10	7	3	5

MPI_Alltoall

int MPI_Alltoall (void **sendbuf*, int *sendcount*, MPI_Datatype *sendtype*,
void **recvbuf*, int *recvcount*, MPI_Datatype *recvtype*,
MPI_Comm *comm*)

sendbuf - start address of send buffer

sendcount - number of elements to send to each process

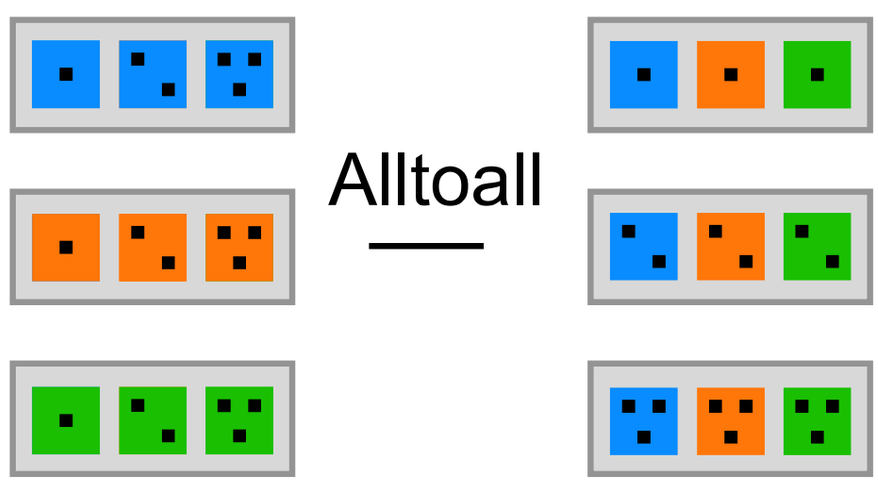
sendtype - datatype of send elements

recvbuf - address of receive buffer

recvcount - number of elements received from any process

recvtype - datatype of recv elements

comm - communicator



In a more general case *sendcount* and *recvcount* will not be just single constants for the whole communication

We want to allow for variable amounts of data to be communicated between each process pair. Therefore must have array describing datatype counts

sendcount -> sendcounts[]
recvcount -> recvcounts[]

As these are now variable, must also have an array describing displacements, as these cannot just be assumed to be multiples of some constant

sdispls[] - send data displacements
rdispls[] - receive data displacements

MPI_Alltoallv

int MPI_Alltoallv (
    void *_sendbuf_, int *_sendcounts_, int *_sdispls_, MPI_Datatype _sendtype_,
    void *_recvbuf_, int *_recvcounts_, int *_rdispls_, MPI_Datatype _recvtype_,
    MPI_Comm _comm_)

sendbuf - starting address of send buffer

sendcounts - array of send counts

sdispls - array of send displacements

sendtype - send datatype

recvbuf - starting address of receive buffer

recvcounts - array of receive counts

rdispls - array of receive displacements

recvtype - receive datatype

comm - communicator

To distribute the keys:

1. Determine what and how much data is to be sent to each process

2. Carry out a total exchange on the amount of data to be sent/received by each process (MPI_Alltoall)

3. Compute the total amount of space needed for the data to be received and allocate storage

4. Find the displacements of the data to be received

5. Carry out a total exchange of the actual keys (MPI_Alltoallv)

6. Free old storage

Modify structure, add pointer which will eventually point to list array.
We also add some useful macros.

```
typedef int KEY_T;
typedef struct {
    int allocated_size;
    int local_list_size;
    KEY_T* keys;
} LOCAL_LIST_T;
```

```
#define List_size(list) ((list)->local_list_size)
#define List_allocated_size(list) ((list)->allocated_size)
#define List(list) ((list)->keys)
#define List_free(list) {free List(list);}
#define List_key(list,i) (*((list)->keys + i))

#define key_mpi_t MPI_INT
```

```
void Redistribute_keys(
        LOCAL_LIST_T* local_keys  /* in/out */) {
    int new_list_size, i, error = 0;
    int* send_counts;
    int* send_displacements;
    int* recv_counts;
    int* recv_displacements;
    KEY_T* new_keys;

    /* Allocate space for the counts and displacements */
    send_counts = (int*) malloc(p*sizeof(int));
    send_displacements = (int*) malloc(p*sizeof(int));
    recv_counts = (int*) malloc(p*sizeof(int));
    recv_displacements = (int*) malloc(p*sizeof(int));

    Local_sort(local_keys);
    Find_alltoall_send_params(local_keys,
        send_counts, send_displacements);

    /* Distribute the counts */
    MPI_Alltoall(send_counts, 1, MPI_INT, recv_counts,
        1, MPI_INT, MPI_COMM_WORLD);
```

```
/* Allocate space for new list */
new_list_size = recv_counts[0];
for (i = 1; i < p; i++)
    new_list_size += recv_counts[i];
new_keys = (KEY_T*)
    malloc(new_list_size*sizeof(KEY_T));

Find_recv_displacements(recv_counts, recv_displacements);

/* Exchange the keys */
MPI_Alltoallv(List(local_keys), send_counts,
    send_displacements, key_mpi_t, new_keys,
    recv_counts, recv_displacements, key_mpi_t,
    MPI_COMM_WORLD);

    /* Replace old list with new list */
    List_free(local_keys);
    List_allocated_size(local_keys) = new_list_size;
    List_size(local_keys) = new_list_size;
    List(local_keys) = new_keys;

    /* Free temporary storage */
    free(send_counts);
    free(send_displacements);
    free(recv_counts);
    free(recv_displacements);
} /* Redistribute_keys */
```

At this stage, Find_alltoall_send_params and Find_recv_displacements are defined by us as stub functions

Thus the code at this stage will compile, but you cannot yet actually run Redistribute_keys with these functions empty so comment it out

```
void Find_alltoall_send_params(LOCAL_LIST_T* local_keys,
        int send_counts[], int send_displacements[]) {

} /* Find_alltoall_send_params */
```

```
void Find_recv_displacements(int recv_counts[],
        int recv_displacements[]) {
}
```

Write function to sort the list at this stage, use intrinsic qsort

```
/*************************************************************/
void Local_sort(LOCAL_LIST_T* local_keys) {

    qsort(List(local_keys), List_size(local_keys), sizeof(KEY_T),
        (int(*)(const void*, const void*))(Key_compare));
} /* Local_sort */

/*************************************************************/
int Key_compare(const KEY_T* p, const KEY_T* q) {
    if (*p < *q)
        return -1;
    else if (*p == *q)
        return 0;
    else /* *p > *q */
        return 1;

}  /* Key_compare */
```

Fill out allocate list function and insert key function

```
/* Return value negative indicates failure */
int Allocate_list(
        int           list_size  /* in  */,
        LOCAL_LIST_T* local_keys /* out */) {

    List_allocated_size(local_keys) = list_size/p;
    List_size(local_keys) = list_size/p;
    List(local_keys) = (KEY_T*)
        malloc(List_allocated_size(local_keys)*sizeof(KEY_T));
    if (List(local_keys) == (KEY_T*) NULL)
        return -1;
    else
        return 0;
} /* Allocate_list */

void Insert_key(KEY_T key, int i,
    LOCAL_LIST_T* local_keys) {
    List_key(local_keys, i) = key;
} /* Insert_key */
```

Fill out Print_list function as well

```
/*********************************************************************/
void Print_list(MPI_Comm io_comm, LOCAL_LIST_T* local_keys) {
    char list_string[LIST_BUF_SIZE];
    char key_string[MAX_KEY_STRING];
    int  i;

    list_string[0] = '\0';
    for (i = 0; i < List_size(local_keys); i++) {
        sprintf(key_string,"%d ", List_key(local_keys,i));
        strcat(list_string, key_string);
    }
    Cprintf(io_comm,"Contents of the list", "%s", list_string);
} /* Print_list */
```

This completes stage 3, one last stage left

We need to add code to compute send_counts and send_displacements, and we are almost done

To write the functions necessary, we can make use of the fact that our local list will be already sorted before they are called. This means we have the relation

send_displacements[q] = send_displacements[q-1] + send_counts[q-1]

Must have separate case for q=0

Also need cutoff value for each process i.e. given process will not be given keys higher than cutoff

As keys are uniformly distributed between 0 and KEY_MAX

cutoff = (q+1)*(KEY_MAX+1)/p

is the cutoff for process p

```
void Find_alltoall_send_params(
        LOCAL_LIST_T* local_keys         /* in  */,
        int*          send_counts        /* out */,
        int*          send_displacements /* out */) {
    KEY_T cutoff;
    int i, j;

    /* Take care of process 0 */
    j = 0;
    send_displacements[0] = 0;
    send_counts[0] = 0;
    cutoff = Find_cutoff(0);
    /* Key_compare > 0 if cutoff > key */
    while ((j < List_size(local_keys)) &&
           (Key_compare(&cutoff,&List_key(local_keys,j))
                > 0)) {
        send_counts[0]++;
        j++;
    }
```

```
/* Now deal with the remaining processes */
for (i = 1; i < p; i++) {
    send_displacements[i] =
        send_displacements[i-1] + send_counts[i-1];
    send_counts[i] = 0;
    cutoff = Find_cutoff(i);
    /* Key_compare > 0 if cutoff > key */
    while ((j < List_size(local_keys)) &&
           (Key_compare(&cutoff,&List_key(local_keys,j))
               > 0)) {
           send_counts[i]++;
           j++;
       }
    }
} /* Find_alltoall_send_params */

/*defined as separate function for easy changing later if needed*/
int Find_cutoff(int i) {
    return (i+1)*(KEY_MAX + 1)/p;
}  /* Find_cutoff */
```

Fill in Find_recv_displacements function

```
void Find_recv_displacements(int recv_counts[],
    int recv_displacements[]){
    int i;

    recv_displacements[0] = 0;
    for (i = 1; i < p; i++)
        recv_displacements[i] =
            recv_displacements[i-1]+recv_counts[i-1];
} /* Find_recv_displacements */
```

Final piece is just a function which will read the only input parameter necessary at runtime, which specifies how big the list is.

Assuming you have working access to standard input:

```
/*****************************************************************/
int Get_list_size(void) {
    int size;

    Cscanf(io_comm,"How big is the list?","%d",&size);
    return size;
} /* Get_list_size */
```

At this point you might consider writing a serial driver for Find_alltoall_send_params to test it out

This driver would read in *p*, *my_rank*, and a list of keys. It would then calculate the send counts and displacements and print them out.

It is worthwhile to write your code explicitly in stages.

There are tools to help you managing your code as you modify it. A good one is called GIT.

Designing Parallel Algorithms

Outline

Methodical design:

- Partitioning
- Communication
- Conglomeration
- Mapping

Methodical design

Partitioning

Computation and data are decomposed into small tasks

Focus is on recognizing opportunities for parallel execution

Communication

Communication requirements, structures, and algorithms are determined

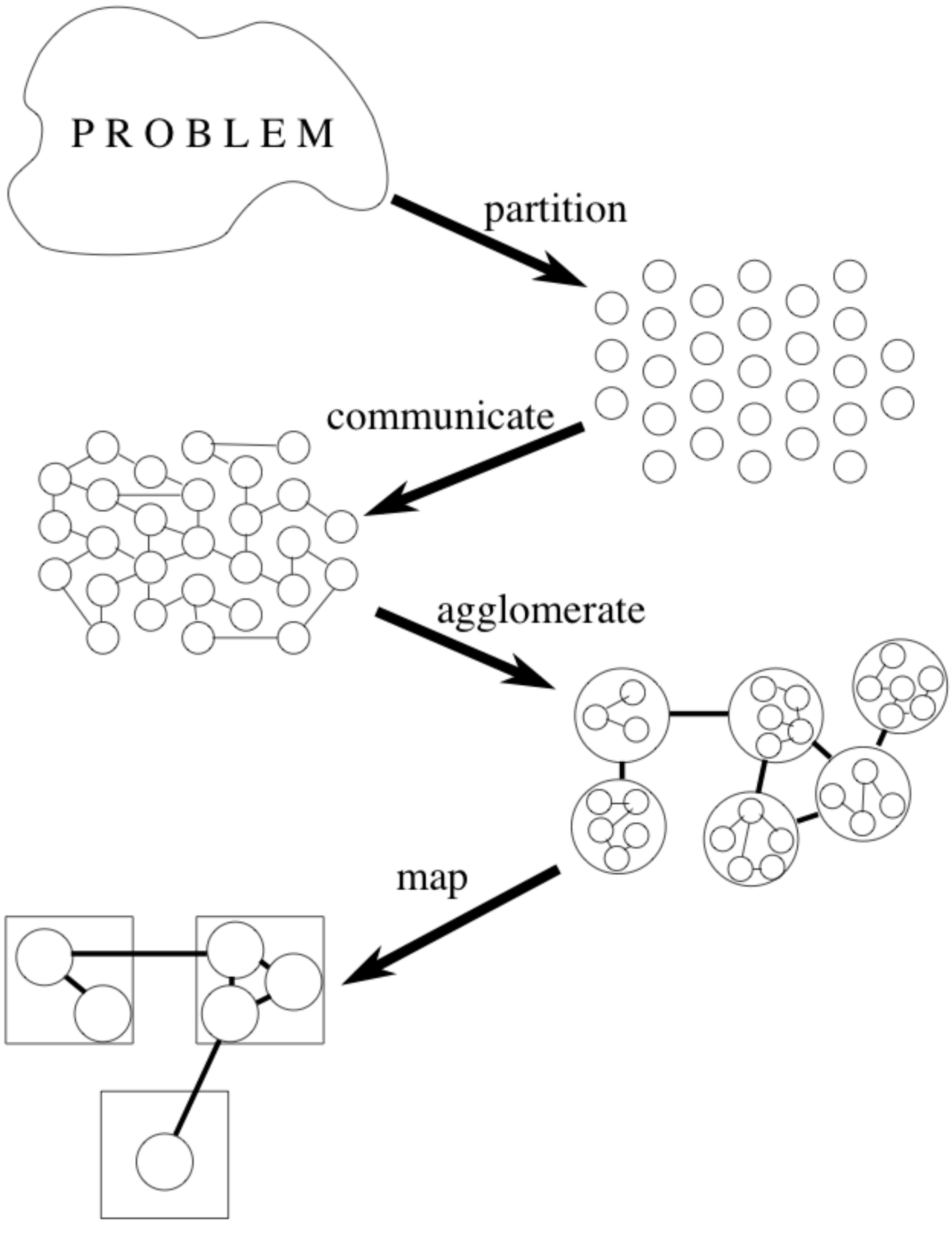
Agglomeration

Tasks are combined into larger tasks to improve performance

Mapping

Each task is assigned to a processor

Goal is to maximize processor utilization and minimize communication costs


PROBLEM
partition
communicate
agglomerate
map

Partitioning

Focus is on recognizing opportunities for parallel execution

Computation and the data operated on by this computation are decomposed

Define a number of small tasks in order to yield what is termed a fine-grained decomposition of a problem

A good partition divides into pieces both computation and data

We try to avoid replicating computation and data

That is, we seek to define tasks that partition both computation and data into disjoint sets

Domain decomposition

- partition the data

- determine communication associated with the partition

Functional decomposition

- decompose the computation into tasks

- decompose the data based on these tasks

Common in problems where there are no obvious data structures to partition

Partition design checklist

Does your partition define at least an order of magnitude more tasks than there are processors in your target computer?

If not, you may have little flexibility in subsequent design stages

Does your partition avoid redundant computation and storage requirements?

If not, it may not be scalable

Are tasks of comparable size?

Does the number of tasks scale with problem size?

Have you identified several alternative partitions?

Communication

Tasks generated by a partition are intended to execute concurrently

The computation to be performed in one task will typically require data associated with another task

Data must then be transferred between tasks, to allow computation to proceed

This information flow is specified in the communication phase of a design

Local communication: each task communicates with a small set of other tasks (its “neighbors”)

Global communication: requires each task to communicate with many tasks

Structured communication: a task and its neighbors form a regular structure, e.g. a tree or grid

Unstructured communication: communication pattern may be complex and irregular

Static communication: the communication pattern does not change over time

Dynamic communication: the communication pattern is determined at runtime and may be variable

Example: Jacobi vs. Gauss-Seidel

Consider Jacobi's method

A multidimensional grid is repeatedly updated by replacing the value at each point with some function of the values at a small, fixed number of neighboring points

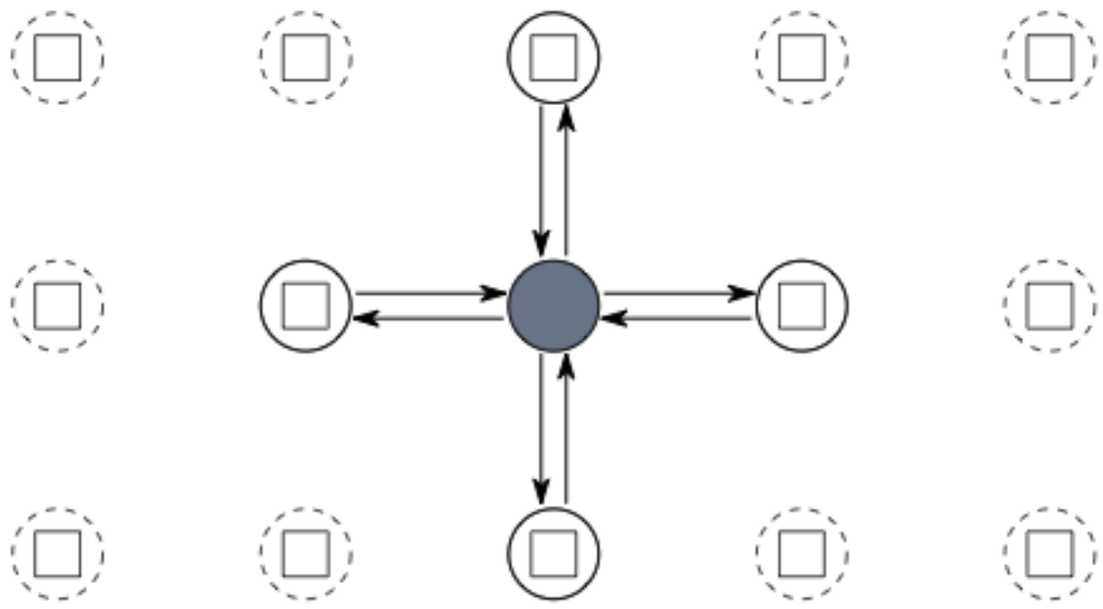
The set of values required to update a single grid point is called that grid point's stencil

For example, the following expression uses a five-point stencil to update each element of a two-dimensional grid:

$$X_{i,j}^{(k+1)} = \frac{4X_{i,j}^{(k)} + X_{i-1,j}^{(k)} + X_{i+1,j}^{(k)} + X_{i,j-1}^{(k)} + X_{i,j+1}^{(k)}}{8}$$

k is iteration number, assume boundary points fixed

$$X_{i,j}^{(k+1)} = \frac{4X_{i,j}^{(k)} + X_{i-1,j}^{(k)} + X_{i+1,j}^{(k)} + X_{i,j-1}^{(k)} + X_{i,j+1}^{(k)}}{8}$$



For each boundary point we have, for k=0,1,...

send $X_{i,j}^{(k)}$ to each neighbor
receive $X_{i-1,j}^{(k)}$, $X_{i+1,j}^{(k)}$, $X_{i,j-1}^{(k)}$, $X_{i,j+1}^{(k)}$
compute $X_{i,j}^{(k+1)}$

In serial, Gauss-Seidel (G-S) is often more efficient than Jacobi

Also, it reuses the same matrix, instead of dealing with two copies

G-S is more difficult to parallelize

Consider the simple G-S scheme

$$X_{i,j}^{(k+1)} = \frac{4X_{i,j}^{(k)} + X_{i-1,j}^{(k+1)} + X_{i+1,j}^{(k)} + X_{i,j-1}^{(k+1)} + X_{i,j+1}^{(k)}}{8}$$

This is efficient numerically but difficult to parallelize

Red-Black ordering

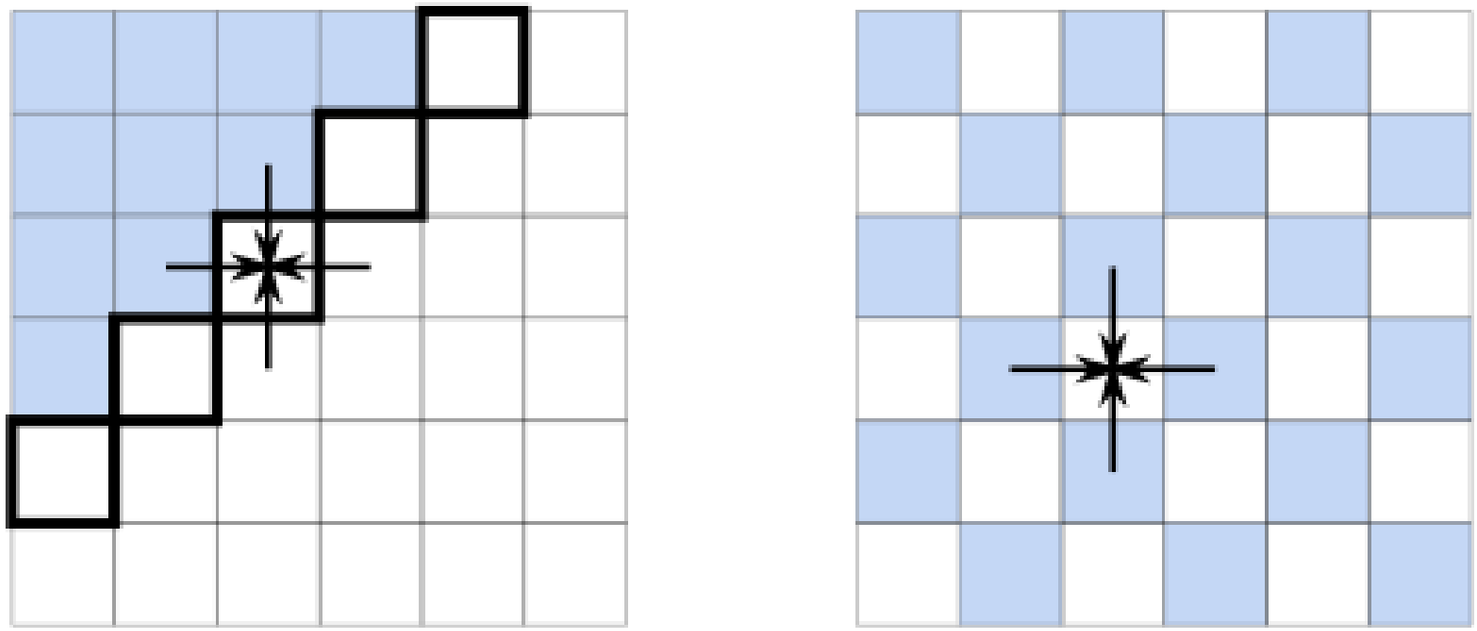
One can update first the odd-numbered elements and then the even-numbered elements of an array

Each update uses the most recent information

Updates to the odd-numbered [resp. even-numbered] points are independent and can proceed concurrently

This is red-black ordering: points can be thought of as being colored as on a chess board, red (odd) or black (even)

Points of the same color can be updated concurrently



Two finite difference update strategies

Shaded grid points have already been updated to step $k + 1$

Unshaded grid points are still at step k

Left: simple G-S; the update proceeds in a wavefront from the top left corner to the bottom right

Right: red-black update scheme; all the grid points at step k can be updated concurrently

The Jacobi update strategy is efficient in parallel, but inferior numerically

The red-black scheme combines the advantages of G-S and Jacobi

Global communication

Global communication: many tasks must participate

One should try to avoid creating too many communications or restricting opportunities for concurrent execution

Example

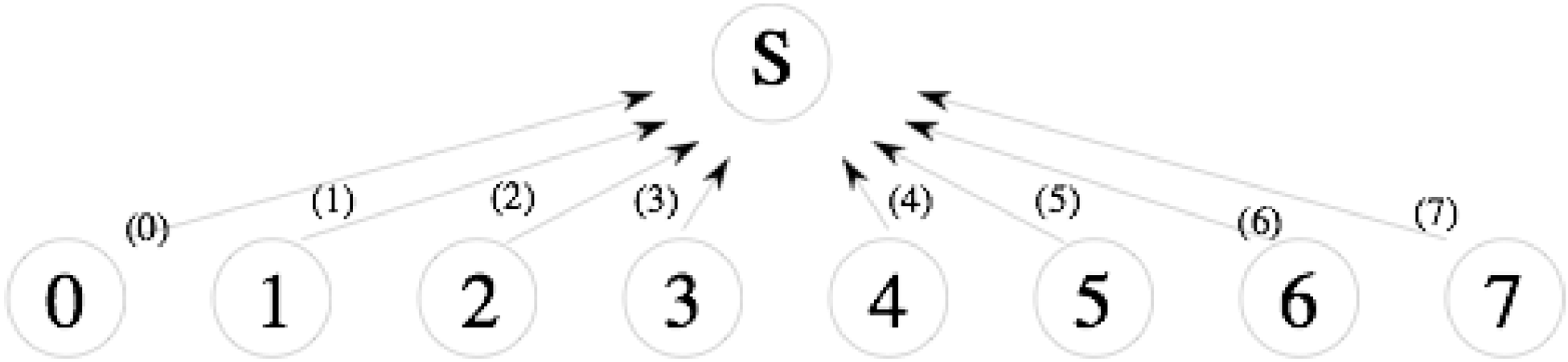
Consider a parallel reduction operation that sums *N* values distributed over *N* tasks

$$S = \sum_{i=0}^{N-1} X_i$$

Assume a single “manager” task requires the result S of this operation

Taking a purely local view of communication, we recognize that the manager requires values X0, X1, etc. from tasks 0,1, etc.

Hence, we could define a communication structure that allows each task to communicate its value to the manager independently



The manager can receive and sum only one number at a time

This approach takes *O(N)* time to sum *N* numbers—not a very good parallel algorithm!

- algorithm is centralized: it does not distribute computation and communication

- algorithm is sequential: it does not allow multiple computation and communication operations to proceed concurrently

We must address both these problems to develop a good parallel algorithm

Distributing communication and computation

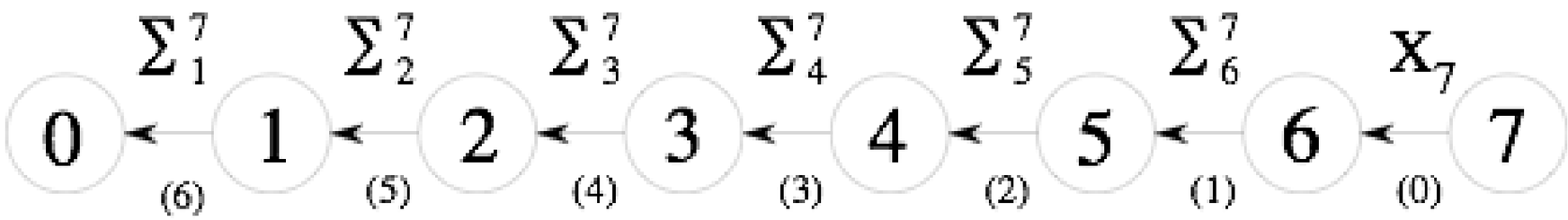
Consider distributing the computation and communication associated with the summation

We can distribute the summation of the N numbers by making each task i, $0 < i < N - 1$, compute the sum:

$$S_i = X_i + S_{i-1}$$

The N tasks are connected in a one-dimensional array

Task N − 1 sends its value to its neighbour in this array



Tasks 1 through $N - 2$ wait to receive a partial sum from their right-hand neighbour, add this to their local value, and send the result to their left-hand neighbour

Task 0 receives a partial sum and adds this to its local value

This approach distributes the $N - 1$ communications and additions, but permits concurrent execution only if multiple summation operations are to be performed

The array of tasks can then be used as a pipeline, through which flow partial sums

A single summation still takes $N - 1$ steps

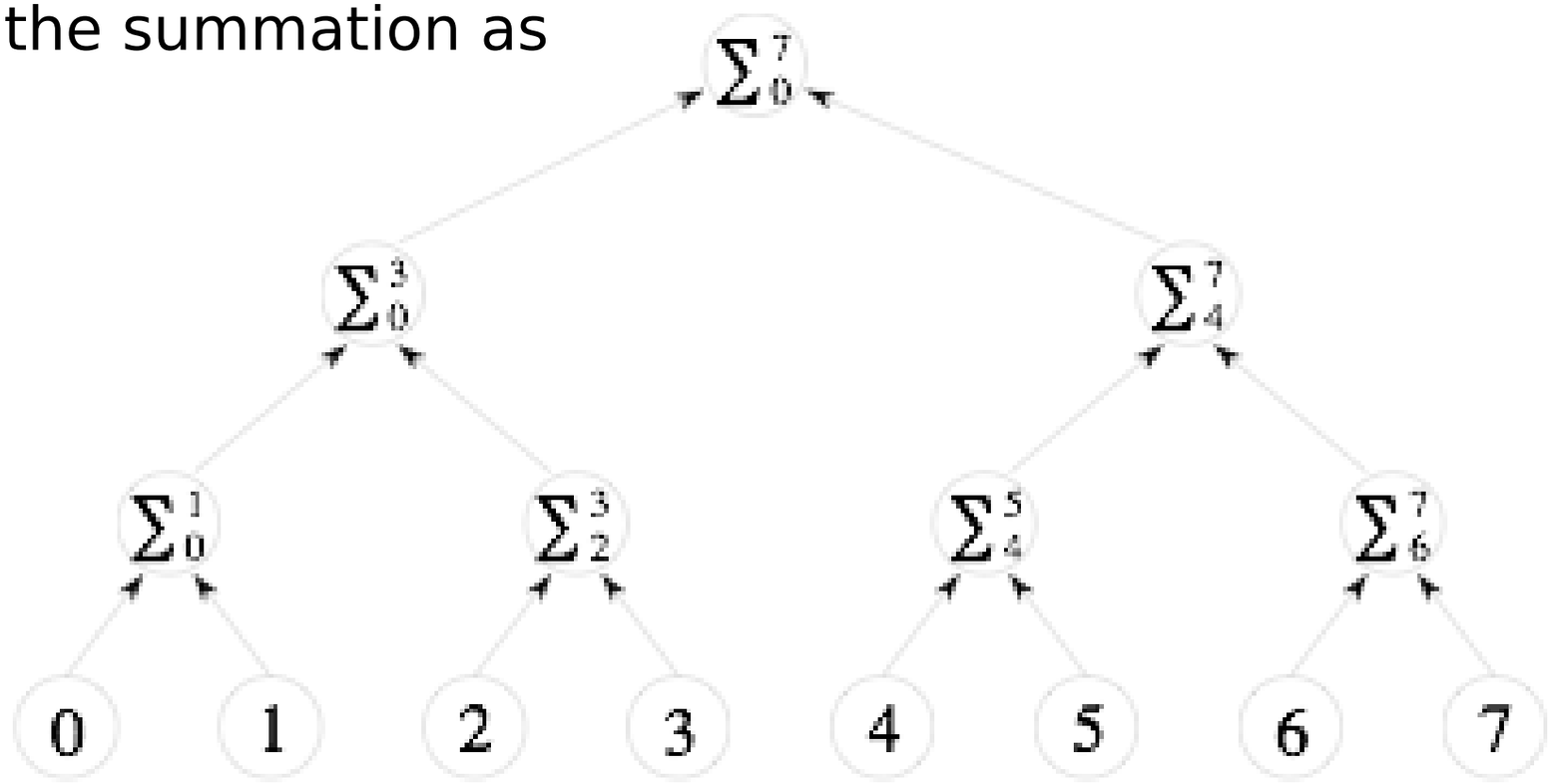
Uncovering concurrency: Divide and Conquer

Divide and conquer: divide into two or more simpler problems of roughly equivalent size (e.g., summing N/2 numbers)

Apply recursively

Effective in parallel computing when the subproblems generated by problem partitioning can be solved concurrently

We can do the summation as



Summations at the same level can be done in parallel

O(log N) communications steps

We distributed $N - 1$ communication and computation operations required to perform the summation

We modified the order in which these operations are performed so that they can proceed concurrently

The result is a regular communication structure in which each task communicates with a small set of neighbours

Unstructured and dynamic communication

The previous examples are all of static, structured communication

In practice communication pattern

- may be considerably more complex and

- may change over time

Communication design checklist

Do all tasks perform about the same number of communication operations?

Unbalanced communication requirements suggest a non-scalable construct
See if communication operations can be distributed more equitably

- Does each task communicate only with a small number of neighbors? If each task must communicate with many other tasks, consider formulating this global communication in terms of local communication

- Are communication operations able to proceed concurrently?

- Is the computation associated with different tasks able to proceed concurrently?

If not, your algorithm is likely to be inefficient and non-scalable

Consider whether you can reorder communication and computation operations

You may also wish to revisit your problem specification (as was done in moving from a simple G-S to a red-black algorithm)

Agglomeration

We move from the abstract toward the concrete

From the first two stages, we have an abstract algorithm; not specialized for efficient execution on any particular parallel computer

It may be highly inefficient if, for example, it creates many more tasks than there are processors on the target computer, and this computer is not designed for efficient execution of small tasks

We revisit decisions made in the partitioning and communication phases to obtain an algorithm that will execute efficiently on some class of parallel computer

We consider whether it is useful to combine, or agglomerate, tasks identified by the partitioning phase, to provide a smaller number of tasks, each of greater size

We also determine whether it is worthwhile to replicate data and/or computation

Increasing granularity

In this case this means making chunks into which algorithm is divided larger

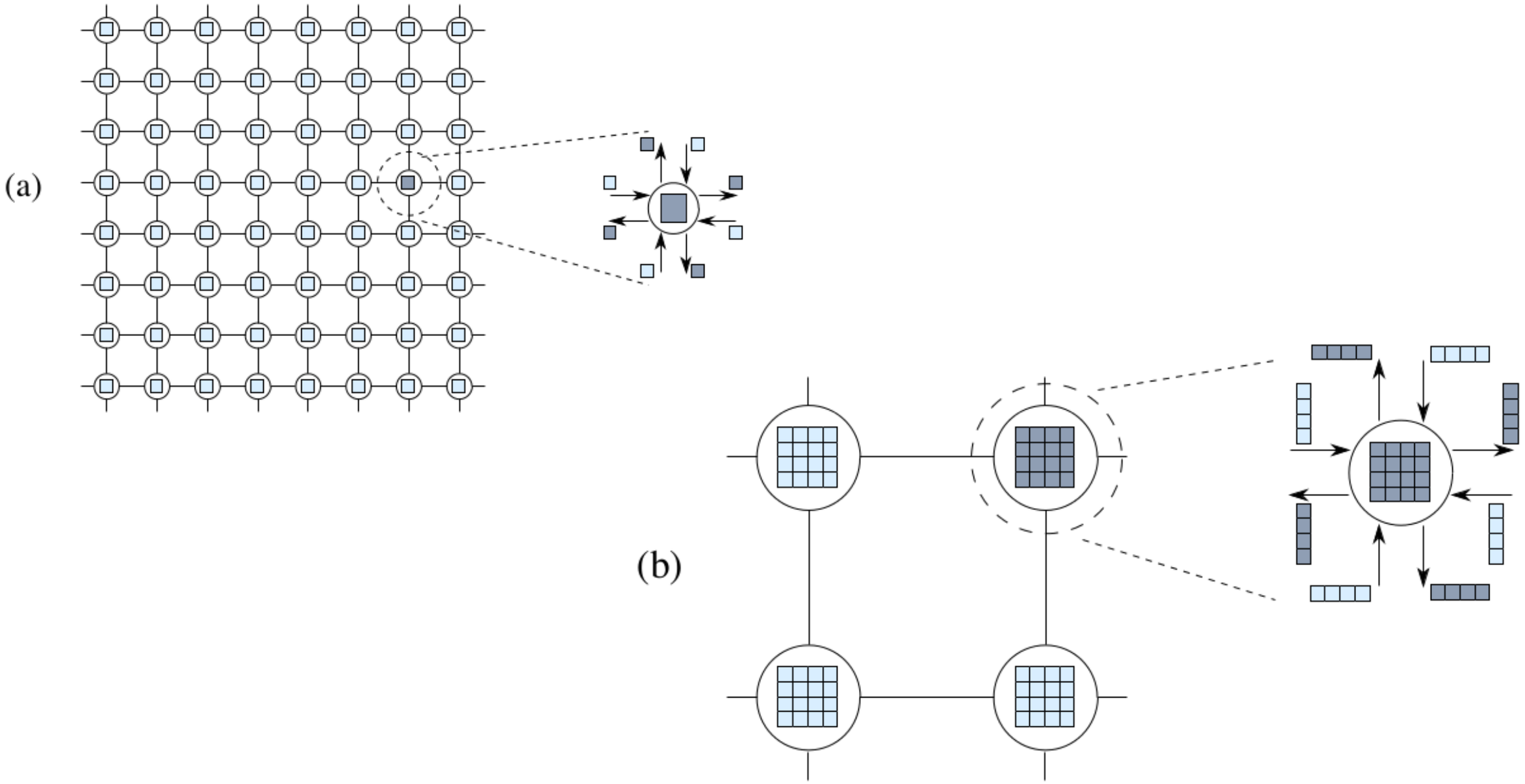
In the partitioning phase, our efforts are focused on defining as many tasks as possible

Forces us to consider a wide range of opportunities for parallel execution

Large number of fine-grained tasks does not necessarily produce an efficient parallel algorithm

Critical issue is communication costs

Each communication incurs not only a cost proportional to the amount of data transferred, but also a fixed startup cost



Effect of increased granularity on communication costs in a two-dimensional finite difference problem, five-point stencil
(a) 64 tasks, each responsible for a single point; 4 communications per task, 64 × 4 = 256 communications, 256 data points transferred
(b) 4 tasks, each responsible for 16 points; 4 × 4 = 16 communications, 16 × 4 = 64 data points transferred

Replicating computation

We can sometimes trade off replicated computation for reduced communication requirements and/or execution time

In other words, you should not necessarily need to insist that everything is computed only once.

Depends a great deal on the particular architecture you are using, how fast communications are etc.

Must be careful about Amdahl’s Law issues

Agglomeration design checklist

(means we are collecting many small tasks into *p* tasks)

Has agglomeration reduced communication costs by increasing locality?

If agglomeration has replicated computation, have you verified that the benefits of this replication outweigh its costs?

If agglomeration replicates data, have you verified that this does not compromise the scalability of your algorithm by restricting the range of problem sizes or processor counts that it can address?

Has agglomeration yielded tasks with similar computation and communication costs?

Does the number of tasks still scale with problem size?

Can the number of tasks be reduced still further, without introducing load imbalances, increasing software engineering costs, or reducing scalability?

Other things being equal, algorithms that create fewer larger-grained tasks are often simpler and more efficient than those that create many fine-grained tasks.

Mapping

We specify where each task is to execute

Two strategies:

1. Place tasks that are able to execute concurrently on different processors, so as to enhance concurrency

2. Place tasks that communicate frequently on the same processor, so as to increase locality.

Resource limitations may restrict the number of tasks that can be placed on a single processor

The mapping problem is known to be NP complete. Goal is to minimize total execution time

Mapping decisions seek to balance conflicting requirements for equitable load distribution and low communication costs

Parallel Libraries

Using Libraries: Pro and Con

Pro:
- Libraries can provide high quality, high performance code
- User does not have to write the parallel program, which is difficult
- MPI has good support for parallel libraries
- Use of communicators allows library to isolate its communications universe from rest of program, avoiding conflicts

Con:
- Writing and documenting a library is difficult and time-consuming, hence some libraries may be deficient in some respects and difficult to use
- Examples of well-designed and well-documented libraries: ScaLAPACK, PETSc

Parallel Libraries: FFTW with MPI

FFTW - Fastest Fourier Transform in the West

Most widely used implementation of FFT (Fast Fourier Transform)

Forward discrete Fourier transform of 1d complex array *X* of size *n*, computes array *Y* of size *n* via:

$$Y_k = \sum_{j=0}^{n-1} X_j e^{-2\pi jk\sqrt{-1}/n}$$

Algorithm scales as *N logN*

Higher dimensional transforms are straightforward extensions of 1D transform

Using FFTW with MPI support

ssh orca
ssh orc-dev1 [2,3,4]
module load fftw

mpicc $CPPFLAGS $LDFLAGS -lfftw3_mpi -lfftw3 test.c

Example: 2D complex forward FFT in parallel

Data distribution: 1D block along first dimension

FFTW will decide how to distribute the data, and this information must be extracted from it and used in the program

Example code from:
http://www.fftw.org/doc/2d-MPI-example.html

```
#include <complex.h>
#include <fftw3-mpi.h>

fftw_complex my_function(ptrdiff_t i_in, ptrdiff_t j_in){
 return 3*i_in+4*j_in*I;  // simple example function
}
```

~syam/ces745/mpi/parallel_fftw/parallel_fftw.c

```
int main(int argc, char **argv)
{
    const ptrdiff_t N0 = 200, N1 = 200;
    fftw_plan plan;
    fftw_complex *data;
    ptrdiff_t alloc_local, local_n0, local_0_start, i, j;

    MPI_Init(&argc, &argv);
    fftw_mpi_init();

    /* get local data size and allocate */
    alloc_local = fftw_mpi_local_size_2d(N0, N1, MPI_COMM_WORLD,
                                         &local_n0, &local_0_start);
    data = fftw_alloc_complex(alloc_local);

    /* create plan for in-place forward DFT */
    plan = fftw_mpi_plan_dft_2d(N0, N1, data, data, MPI_COMM_WORLD,
                                FFTW_FORWARD, FFTW_ESTIMATE);
```

```
/* initialize data to some function my_function(x,y) */

for (i = 0; i < local_n0; ++i)
     {
      for (j = 0; j < N1; ++j)
      {
           data[i*N1 + j] = my_function(local_0_start + i, j);
      }
     }

     /* compute transforms, in-place, as many times as desired */
     fftw_execute(plan);

     fftw_destroy_plan(plan);

     MPI_Finalize();
}
```