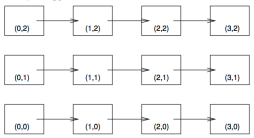
Finite Difference Algorithm

```
int finte_difference() {
      int i, j, n;
      double u[n, n], unew[n, n];
      for (k = 0; k < ITERS; k++) {
            for (i = 0; i < n; i++) {
                  for (j = 0; j < n; j++) {
                        unew[i, j] = 0.25 * (u[i - 1, j] + u[i + 1,
j] + u[i, j - 1] + u[i, j + 1]) - f[i, j];
                  }
            }
            diffmax = 0.0;
            for (i = 1; i < n; i++) {
                  for (j = 1; j < n; j++) {
                        diff = abs(unew[i, j] - u[i, j]);
                        if (diff > diffmax) diffmax = diff;
                        u[i, j] = unew[i, j];
                  }
            }
      }
}
```

Basic Sweep Routine

```
int sweep() {
      int i, j, n;
      double u[n, n], unew[n, n];
      for (i = 0; i < n; j++) {
            for (j = 0; j < n; j++) {
                  unew[i, j] = 0.25 * (u[i - 1, j] + u[i+1, j] +
u[i, j - 1] + u[i, j + 1]) - f[i, j];
            }
      }
}
```

Illustration of Cartesian Topology



Finding Neighbors of a Topology

- To determine the coordinates of a calling process

```
MPI_Cart_get(commId, 2, *dims, *periods, *coords);
printf(‘(’, coords[0], ‘,’ coords[1], ‘)’);
```

- To determine rank

```
MPI_Cart_rank(commId, *coord, *myrank);
printf(myrank);
```

Using Topologies

- Performing shift operations

```
MPI_Cart_shift(commId, direction, shift, *src, *dest);
```

Data Exchange Routine

```
int exchngl(a, nx, commId, nbrbottom, nbrtop) {
      int nx;
      double a(n / P + 2, n);
      int commId, nbrbottom, nbrtop;
      int status(MPI_STATUS_SIZE), ierr;
      if (nbrtop != MPI_UNDEFINED) {
            MPI_Send(&a(1, 0), nx, MPI_DOUBLE, nbrtop, 0, commId);
            MPI_Recv(&a(0, 0), nx, MPI_DOUBLE, nbrtop, 0, commId);
      }
      if (nbrbottom != NPI_UNDEFINED) {
            MPI_Send(&a(n / P, 0), nx, MPI_DOUBLE, nbrbottom, 1,
commId);
            MPI_Recv(&a(n / P + 1, 0), nx, MPI_DOUBLE, nbrbottom,
1, commId);
      }
}
```

Basic Jacobi Sweep Routine

```
int sweepId(&a, &f, nx, &b);
int nx;
double a(n / P + 2, n), f(n / P + 2, n), b(n / P + 2, n);
int i, j;
double h;
h = 1.0 / (nx + 1);
for (i = 1; i < (n / P + 1); i++) {
      for (j = 0; j < n; j++) {
            b[i, j] = 0.25 * (a[i - 1, j] + a[i + 1, j] + a[i, j -
1] + a[i, j + 1]) - h * h * f[i, j];
      }
}
```

Overall Jacobi Implementation

```
main() {
      #define maxn 128;
      double a(maxn, maxn), b(maxn, maxn), f(maxn, maxn);
      int nx, ny;
      int myid, numprocs;
      int commId, nbrbottom, nbrtop, it;
      MPI_Init();
      MPI_Comm_rank(MPI_COMM_WORLD, myid);
      MPI_Comm_size(MPI_COMM_WORLD, numprocs);
      if (myid == 0) {
            nx = 100;
      }
      MPI_Bcast(nx, 1, MPI_INT, 0, MPI_COMM_WORLD);
      ny = nx;
      //Create a one dim cartesian mapping
      MPI_Cart_Create(MPI_COMM_WORLD, 1, numprocs, 0, 1, commId);
      //Get my position in this communicator, and my neighbors
      MPI_Comm_Rank(commId, myid);
      MPI_Cart_shift(commId, 0, 1, &nbrbottom, &nbrtop);
      //Initialize the right-hand-side (f) and the initial
solution guess (a)
      onedinit(a, b, f, nx);
      MPI_Barrier(MPI_COMM_WORLD);
      for (it = 1; it < 100; it++) {
            exchngl(a, nx, commId, nbrbottom, nbrtop);
            sweepId(a, f, nx, b);
            dwork = diff(a, b, nx, s, e);
            MPI_Allreduce(dwork, diffnorm, 1, MPI_DOUBLE, MPI_MAX,
commId);
            if (diffnorm < 1.0e-5) break;
            exchngl(b, nx, commId, nbrbottom, nbrtop);
            sweepId(b, f, nx, a);
            dwork = diff(a, b, nx, s, e);
            MPI_Allreduce(dwork, diffnorm, 1, MPI_DOUBLE, MPI_MAX,
commId);
            if (diffnorm < 1.0e-5) break;
      }
      MPI_Finalize();
}
```

The corresponding parallel program is:

```
int a[25], b[25], c[25];
main() {
      MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs);
      MPI_COMM_RANK(MPI_COMM_WORLD, &id);
      if (id > 0) {
            MPI_SEND(&b[0], 1, MPI_FLOAT, id - 1, 0,
MPI_COMM_WORLD);
      }
      if (id < 3) {
            MPI_RECV(&b[25], 1, MPI_FLOAT, id + 1, 0,
MPI_COMM_WORLD);
      }
      for (i = 0; i < 25; i++) {
            a[i] = b[i + 1] + b[i] * c[i];
      }
}
main() {
      float a[100], b[100];
      int i, p, lb, ub, id, nprocs;
      float x;
      MPI_COMM_RANK(MPI_COMM_WORLD, id);
      MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs);
      if (id == 0) {
            for (p = 1; p < nprocs; p++) {
                  lb = p * 100 / nprocs;
                  MPI_SEND(&(a[lb]), 100 / nprocs, MPI_FLOAT, p,
ATYPE, MPI_COMM_WORLD);
                  MPI_SEND(&(b[lb]), 100 / nprocs, MPI_FLOAT, p,
BTYPE, MPI_COMM_WORLD);
            }
      } else {
            MPI_RECV(a, 100 / nprocs, MPI_FLOAT, 0, ATYPE,
MPI_COMM_WORLD);
            MPI_RECV(b, 100 / nprocs, MPI_FLOAT, 0, BTYPE,
MPI_COMM_WORLD);
      }
      for (i = 0; i < 100; i++) {
            x = a[i] * 3;
            b[i] = x * b[i];
      }
      if (id == 0) {
            for (p = 1; p < nprocs; p++) {
                  lb = p * 100 / nprocs;
                  MPI_RECV(&(b[lb]), 100 / nprocs, MPI_FLOAT, p,
BTYPE, MPI_COMM_WORLD);
            }
      } else {
            MPI_SEND(b, 100 / nprocs, MPI_FLOAT, 0, BTYPE,
MPI_COMM_WORLD);
      }
}
mian() {
      float a[100], b[10000];
      int i, i1, id, nprocs, p, lb;
      MPI_COMM_RANK(MPI_COMM_WORLD, id);
      MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs);
      if (id == 0) {
            for (p = 1; p < nprocs; p++) {
                  MPI_SEND(&b, 10000, MPI_FLOAT, p, BTYPE,
MPI_COMM_WORLD);
            }
      } else {
            MPI_RECV(&b, 10000, MPI_FLOAT, 0, BTYPE,
MPI_COMM_WORLD);
      }

      offset = id * (100 / nprocs);
      for (i = 0; i < 100 / nprocs; i++) {
            i1 = i + offset;
            indx = ((i1 * (i1 + 1)) / 2;
            a[i] = b[indx];
      }
      if (id == 0) {
            for (p = 1; p < nprocs; p++) {
                  lb = p * 100 / nprocs;
                  MPI_RECV(&(a[lb]), 100 / nprocs, MPI_FLOAT, p,
ATYPE, MPI_COMM_WORLD);
            }
      } else {
            MPI_SEND(a, 100 / nprocs, MPI_FLOAT, 0, ATYPE,
MPI_COMM_WORLD);
      }
}
main() {
      float a[100], total;
      int i, id, nprocs, p, lb;
      float sub_total;
      MPI_COMM_RANK(MPI_COMM_WORLD, id);
      MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs);
      if (id == 0) {
            total = 0.0;
            for (p = 1; p < nprocs; p++) {
                  lb = p * 100 / nprocs;
                  MPI_SEND(&(a[lb], 100 / nprocs, MPI_FLOAT, p,
ATYPE, MPI_COMM_WORLD);
            }
      } else {
            MPI_RECV(&a, 100 / nprocs, MPI_FLOAT, 0, ATYPE,
MPI_COMM_WORLD);
      }
      sub_total = 0.0;
      for (i = 0; i < 100 / nprocs; i++) {
            sub_total = sub_total + a[i];
      }
      MPI_REDUCE(&sub_total, &total, 1, ATYPE, MPI_SUM, 0,
MPI_COMM_WORLD);
}
```

Prescheduling

```
MPI_INIT();
MPI_COMM_RANK(comm, id);
if (id == 0) {
      for (i = 0; i < 30; i++) {
            a[i] = b[i] * c[i];
      }
      //send data for iterations 30 to 69 to node 1
      MPI_SEND(&b[30], 40, MPI_INT, 1, BTYPE, MPI_COMM_WORLD);
      MPI_SEND(&c[30], 40, MPI_INT, 1, CTYPE, MPI_COMM_WORLD);
      //send data for iterations 70 to 99 to node 2
      MPI_SEMD(&b[70], 30, MPI_INT, 2, BTYPE, MPI_COMM_WORLD);
      MPI_SEND(&c[70], 30, MPI_INT, 2, CTYPE, MPI_COMM_WORLD);
      //receives result for iterations 30 to 69 from node 1
      MPI_RECV(&a[30], 40, MPI_INT, 1, ATYPE, MPI_COMM_WORLD);
      //receives result for iterations 70 to 99 from node 2
      MPI_RECV(&a[70], 30, MPI_INT, 2, ATYPE, MPI_COMM_WORLD);
}
if (id == 1) {
      MPI_RECV(&b, 40, MPI_INT, 0, BTYPE, MPI_COMM_WORLD);
      MPI_RECV(&c, 40, MPI_INT, 0, CTYPE, MPI_COMM_WORLD);
      for (i = 0; i < 40; i++) {
            a[i] = b[i] * c[i];
      }
      MPI_SEND(&a, 30, MPI_INT, 0, ATYPE, MPI_COMM_WORLD);
}
```

Static Blocked Scheduling

```
MPI_COMM_RANK(MPI_COMM_WORLD, id);
MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs);
if (id == 0) {
      for (i = 0; i < 100 / nprocs; i++) {
            a[i] = b[i] * c[i];
      }
      for (p = 1; p <= nprocs; p++) [
            lb = p * 100 / nprocs;
            ub = (p + 1) * 100 / nprocs;
            MPI_SEND(&b[lb], 100 / nprocs, MPI_INT, p, BTYPE,
MPI_COMM_WORLD);
            MPI_SEND(&c[lb], 100 / nprocs, MPI_INT, p, CTYPE,
MPI_COMM_WORLD);
      }
      for (p = 1; p <= nprocs; p++) {
            lb = p * 100 / nprocs;
            ub = (p + 1) * 100 / nprocs;
            MPI_RECV(&a[lb], 100 / nprocs, MPI_INT, p, ATYPE,
MPI_COMM_WORLD);
      }
} else {
      MPI_RECV(&b, 100 / nprocs, MPI_INT, 0, BTYPE,
MPI_COMM_WORLD);
      MPI_RECV(&c, 100 / nprocs, MPI_INT, 0, CTYPE,
MPI_COMM_WORLD);
      for (i = 0; i < 100 / nprocs; i++) {
            a[i] = b[i] * c[i];
      }
      MPI_SEND(&a, 100 / nprocs, MPI_INT, 0, ATYPE,
MPI_COMM_WORLD);
}
```

Alternate Form: Static Blocked Scheduling

```
MPI_COMM_RANK(MPI_COMM_WORLD, id);
MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs);
MPI_SCATTER(&b, 100 / nprocs, MPI_INT, &btmp, 100 / nprocs,
MPI_INT, 0, MPI_COMM_WORLD);
MPI_SCATTER(&c, 100 / nprocs, MPI_INT, &ctmp, 100 / nprocs,
MPI_INT, 0, MPI_COMM_WORLD);
for (i = 0; i < 100 / nprocs; i++) {
      atmp[i] = btmp[i] * ctmp[i];
}
MPI_GATHER(&atmp, 100 / nprocs, MPI_INT, &a, 100 / nprocs,
MPI_INT, 0, MPI_COMM_WORLD);
```

Dynamic Scheduling

```
MPI_COMM_RANK(MPI_COMM_WORLD, id);
MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs);
chunk = 10;
if (id == 0) {
      thereiswork = TRUE;
      global_i = 0;
      while (thereiswork) {
            for (p = 1; p <= nprocs; p++) [
                  if (global_i > 100) {
                        thereiswork = FALSE;
                  }
                  MPI_SEND(&thereiswork, 1, MPI_INT, p, TERMTYPE,
MPI_COMM_WORLD);
                  lb = global_i;
                  ub = global_i + chunk;
                  global_i = global_i + chunk;

                  MPI_SEND(&b[lb], chunk, MPI_INT, p, BTYPE,
MPI_COMM_WORLD);
                  MPI_SEND(&c[lb], chunk, MPI_INT, p, CTYPE,
MPI_COMM_WORLD);
                  MPI_RECV(&a[lb], chunk, MPI_INT, p, ATYPE,
MPI_COMM_WORLD);
            }
      }
} else {
      thereiswork = TRUE;
      while (thereiswork) {
            MPI_RECV(&thereiswork, 1, MPI_INT, 0, TERMTYPE,
MPI_COMM_WORLD);
            MPI_RECV(&b, chunk, MPI_INT, 0, BTYPE,
MPI_COMM_WORLD);
            MPI_RECV(&c, chunk, MPI_INT, 0, CTYPE,
MPI_COMM_WORLD);

            for (i = 0; i < chunk; i++) [
                  a[i] = b[i] * c[i];
            }
            MPI_SEND(&a, chunk, MPI_INT, 0, ATYPE,
MPI_COMM_WORLD);
      }
}
```

Example Application: Computation of PI

```
#include "mpi.h"
#include <math.h>
double f(a)
double a;
{
      return (4.0 / (1.0 + a * a));
}
int main(argc, argv)
int argc;
char *argv[];
{
      int n, myid, numprocs, i, rc;
      double PI25DT = 3.1415926
      double mypi, pi, h, sum, x, a;
      double startwtime, endwtime;
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);
      n = 0;
      if (myid == 0) {
            printf("Enter the number of intervals: (0 quits)");
            scanf("%d", &n);
            if (n == 0) {
                  n = 100;
            }
            startwtime = MPI_Wtime();
      }
      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      h = 1.0 / (double) n;
      sum = 0.0;
      for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double) i - 0.5);
            sum += f(x);
      }
      mypi = h * sum;
      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
      if (myid == 0) {
            printf("pi is approximately %.16f\n", pi, fabs(pi -
PI25DT));
            endwtime = MPI_Wtime();
            printf("wall clock time = %f\n", endwtime -
startwtime);
      }
      MPI_Finalize();
}
```

Partitioning and Communication deal with machine independent issues and affect concurrency and scalability
Agglomeration and Mapping deal with machine dependent issues and affect locality and other performance issues

Partitioning Checklist
Does your partition define at least an order of magnitude more tasks that there are processors?
Does your partition avoid redundant computation and storage requirements?
Are tasks of equal size?
Does number of tasks scale with problem size?
Have you identified alternate partitions?

Communications
Tasks generated by above partitioning intended to execute concurrently, but not independently
The computation to be performed in one task requires data from another
Information flow is specified in communication phase
We can conceptualize a need for communication between two tasks as a channel linking tasks (message passing)
Above is easy for functional decomposition, not easy for domain decomposition
Local versus global
Local communication: each task communicates with small set of other tasks
Global communication: each task communicates with many tasks
Structured versus unstructured
Structured communication: a task and its neighbors form a regular structure: tree, grid
Unstructured communication: arbitrary graphs
Static versus dynamic
Static: identity of communication partners does not change
Dynamic: communication partners determined at runtime, data dependent
Synchronous versus asynchronous
Synchronous: Producers and consumers execute in coordinated fashion
Asynchronous: may require consumer obtain data without cooperation of producer

Communication Design Checklist
Do all tasks perform same amount of communication operations?
Dose each task communicate with small number of neighbors?
Are communication operations able to proceed concurrently?
Is computation for different tasks able to proceed concurrently?

Agglomeration
In first two phases, we partitioned the computation to be performed into a set of tasks and introduced communication to provide data required by these tasks
In agglomerate stage, we consider if it is useful to combine tasks to provide smaller number of tasks
Determine if worthwhile to replicate data or computation

Increasing Granularity
In partitioning phase, efforts focused on exposing parallelism
Large number of parallel tasks does not result in efficient parallel algorithm
For communication, send same data in less number of messages
Also, less task creation costs and task scheduling costs

Replicating Computations
Consider problem of replicating the sum of N numbers in N processors
Use a sum reduction followed by a broadcast takes 2(N-1) in ring and 2logN for a tree
Can accomplish in logN steps in butterfly algorithm

Agglomeration Checklist
Has agglomeration reduced communication costs by increasing locality?
If agglomeration uses replicated computations, do benefits outweigh costs?
If agglomeration replicates data, is scalability affected?
Has agglomeration yielded tasks of similar computation and communication costs?
Does number of tasks still scale with problem size?

Mapping
Final stage of parallel algorithm design
Specify which processor each task will execute
Place tasks that are able to execute concurrently on different processors
Place tasks that communicate frequently to same processor
These are conflicting goals

Mapping Problem
Mapping problem to minimize execution time is NP-complete, i.e. no polynomial time algorithm exists for optimal solution
Hence resort to heuristics
Many algorithms developed using domain decomposition techniques feature a fixed number of equal sized tasks and structured and regular communication
Then mapping is straightforward
In more complex domain decomposition based algorithms with variable amount of work per task and unstructured communication, difficult to do mapping

Load Balancing Strategies
In computing, load balancing improves the distribution of workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units, or disk drives. Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource. Using multiple components with load balancing instead of a single component may increase reliability and availability through redundancy. Load balancing usually involves dedicated software or hardware.
Time required to execute these algorithms weighed against benefits of reduced execution costs
Dynamic load balancing: where a load balancing algorithm is periodically executed to determine a new mapping
Local versus global algorithms
Probabilistic versus deterministic

Recursive Bisection Load Balancing
Recursive bisection techniques are used to partition a domain (e.g. finite element grid) into subdomain of approximately equal computational cost
Attempt to minimize communication costs
A divide and conquer approach is taken
Most straightforward approach is recursive coordinate bisection
Makes cuts based on physical coordinates of domain
At each step, subdivide along longer dimension (say x) so that at that step, points in one subdomain will have all x-coordinates greater than grid points in the other
Good job of partitioning computations equally, but does not take care of communication

Other Recursive Bisection Techniques
Unbalanced Recursive Bisection
Recursive Graph Bisection
Recursive Spectral Bisection

Local Load Balancing Algorithms
Previous techniques are expensive since they require global knowledge of computation state
Local algorithms compensate for changes in computational load using information from small number of neighboring processors
E.g. if processors arranged as a logical mesh, each processor compares load with that of its neighbors and transfers computations if the difference in load exceeds some threshold

Probabilistic Load Balancing
Simple load balancing method
Allocate tasks randomly
If number of tasks is large, scheme works well
Advantage: low cost, scalable
Disadvantage: off-processor communication required for almost every tasks

Cyclic Mappings
Cyclic of scattered mapping
Each of P processors is assigned every Pth task

Task Scheduling Algorithms
When a functional decomposition is used, need task scheduling
Conflicting requirements of:
Independent operations to reduce communication costs
Global knowledge of computation state to improve load balance
Manager worker Schemes
Decentralized Schemes

Manager-Worker Scheme
Workers repeatedly request and process problem descriptions
Manager maintains a pool of problem descriptions

Decentralized Schemes
In completely decentralized schemes, there is no central manager
A separate task pool is maintained on each processor
Idle workers request problems from other processors
Task queue becomes distributed data structure accessed by different processors in asynchronous fashion
Variations: a worker may request work from a small number of predefined neighbors or may select other processors at random

Mapping Design Checklist
If considering an SPMD design for a complex problem, have you considered an algorithm based on dynamic task creation, and vice versa?
If using a centralized load-balancing algorithm, have you verified that manager is not bottleneck?
Within dynamic load balancing algorithms, have you evaluated costs of different strategies?