HPC & PARALLELIZATION

TUSAIL School, Linz, Austria, April 27, 2023

Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC)

`Wolfgang.Schreiner@risc.jku.at`

`http://www.risc.jku.at`

JOHANNES KEPLER
UNIVERSITY LINZ

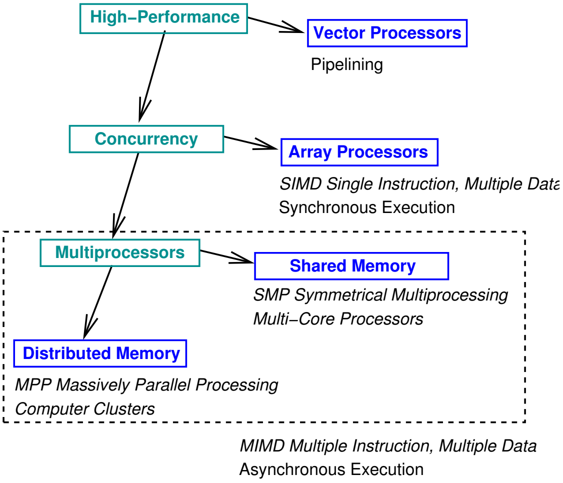
1. HPC Architectures

2. OpenMP

3. MPI

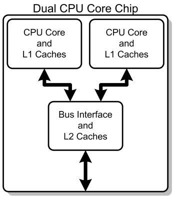
4. Parallel Program Design

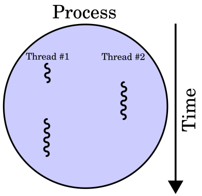
High-Performance Architectures



Architectures that apply concurrency to speedup computations.

Multi-Core Processors



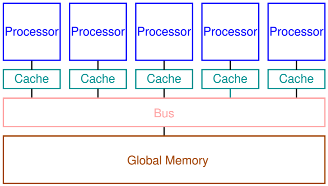


Multi-core processor, *Thread*, en.wikipedia.org

- Processors hold multiple processing units (“cores”).
    - Each core has a separate Level 1 cache.
    - Cores share a common Level 2 cache.
- Cores may execute multiple threads independently.
    - Threads: light-weight processes that can be independently scheduled for execution.
    - Processes: containers that hold multiple threads that have access to the same memory.

Already a processor is a HPC system with concurrent units.

Shared Memory Multi-Processors

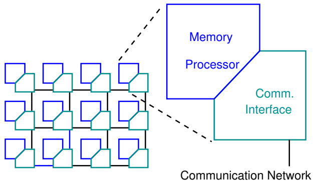


Alternative Term: SMP (Symmetric Multiprocessing)

- Multiple asynchronously operating processors.
- Single OS image schedules processes to processors.
- Single shared memory accessible via central bus.
    - Only one processor at a time can read/write memory.
    - Processors connected to bus via *coherent* caches.
    - *Snooping protocol:* whenever a cache sees another processor's write, it updates its local cache copies.

Scalable to 16 processsors or so.

Distributed Memory Multi-Processors



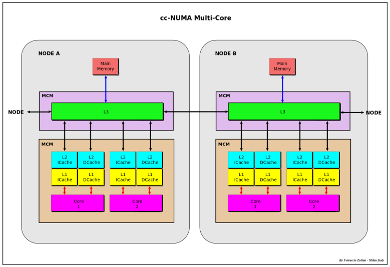
Alternative Term: MPP (Massively Parallel Processing)

- Many identical nodes that operate asynchronously.
  - Processor, local memory, communication interface.
- Each node runs its own OS image.
  - New processes are scheduled to the local processor.
- Nodes connected by high-bandwidth/low-latency network.
  - Different topologies (grid, tree, hypercube, . . .).
  - Different network technologies (InfiniBand, OmniPath, . . .)
  - Remote processes can communicate by *message passing*.

Scalable to thousands of processors.

Virtual Shared Memory Multi-Processors

- ccNUMA: “cache coherent non-uniform memory access”.
    - All local memories combined to single address space.
    - NUMA: access to remote memory is more expensive.
    - Directory keeps track of which nodes hold cache copies of which lines of local memory.
    - If local memory line is updated, nodes with copies are informed.



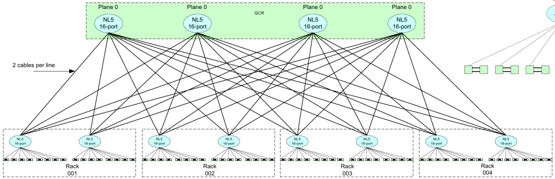
Cache memory, en.wikipedia.org

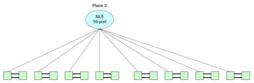
Implementation of SMP model on top of MPP hardware.

The JKU Supercomputer “Mach”

SGI UV-1000 an der Johannes Kepler Universität

Rechnerarchitektur	SGI UV-1000 shared Memory/cc-numa Architektur
Prozessortyp	Intel E78837 (Westmere - EX) X86-64, 2.66GHz / 8-Cores / 24MB Cache
Prozessoranzahl	256 (2048 Cores)
Speicher	16 TB shared Memory
Betriebssystem	Linux – Suse SLES 11 mit SGI Performance Suite
Prozessorleistung	gesamt Peak = 21,3 TFlops Spec_2006_INT Rate = ~39.000 Spec_2006_FP Rate = ~29.000 Stream = 5,8 Tbyte/s Linpack 100 = ~2,2 Gflop/s Linpack NxN = 18,5 Tflop/s
Memory-Bandbreite	7,5 TB / s
Bisection-Bandbreite	480 GB / s





See https://www.risc.jku.at/projects/mach2 for Mach-2.

The Data Access Hierarchy

Data Hierarchy Layer	Latency	Normalized Access Time
L1 Cache	1.4 ns	1$\times$
L3 Cache	23 ns	16$\times$
Local Memory	75 ns	53$\times$
Remote Memory	1 μs	700$\times$
Disk	2 ms	$3.6 \cdot 10^6 \times$

Processors	Cores	Router Hops
2	16	0
32	256	1
256	2048	3

Considering the placement of processes and data is important for achieving high performance on a NUMA system.

1. HPC Architectures

2. OpenMP

3. MPI

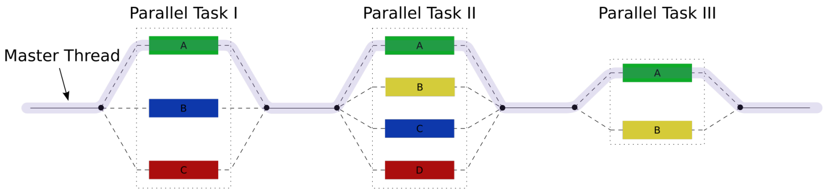
4. Parallel Program Design

OpenMP (OMP)

- An API for portable shared memory parallel programming.
  - Compiler directives (pragmas), library routines, environment variables.
- Targets are C, C++, Fortran.
  - Often used in combination with MPI (Message Passing Interface) for hybrid MPP/SMP programs.
- Widely supported.
  - Commercial compilers: Intel, IBM, Oracle, ...
  - Free compilers: GCC, Clang.
- Maintained by the OpenMP ARB.
  - Architecture Review Board.
  - Current Version: OpenMP 5.2 (November 2021).

See `http://openmp.org` for the official specification.

Programming Model



en.wikipedia.org, *OpenMP*

- Master thread executes program in sequential mode.
- Reaches code section marked with OMP directive:
    - Execution of section is distributed among multiple threads.
    - Main thread waits for completion of all threads.
    - Execution is continued by main thread only.

A fork-join model of parallel execution.

Shared versus Private Variables

The default context of a variable is determined by some rules.

- Static variables and heap-allocated data are shared.
- Automatically allocated variables are
  - Shared, when declared outside a parallel region.
  - Private, when declared inside a parallel region.
- Loop iteration variables are private within their loops.
  - After the loop, the variable has the same value as if the loop would have been executed sequentially.
- . . .

OpenMP clauses may specify the context of variables directly.

Controlling the Number of Threads

- Default set by environment variables:

```
export OMP_DYNAMIC=FALSE
export OMP_NUM_THREADS=4
```

- May be overridden for all subsequent code sections:

```
omp_set_dynamic(0);
omp_set_num_threads(4);
```

- May be overridden for specific sections:

```
#pragma omp parallel ... num_threads(4)
```

If dynamic adjustement is switched on, the actual number of threads executing a section may be smaller than specified.

Controlling the Affinity of Threads to Cores

- Pin threads to cores:

```
export OMP_PROC_BIND=TRUE
```

- Specify the cores (GCC, Intel Compilers):

```
export GOMP_CPU_AFFINITY="256-271" // 16 physical cores in upper half
```

- More flexible alternative for Intel compilers:

```
  export KMP_AFFINITY=
    "verbose,granularity=core,explicit,proclist=[256-271]"
```

Compiling and Executing OpenMP

- Source

```
#include <omp.h>
```

- Intel Compiler:

```
module load intelcompiler/composer_xe_2013.4.183
icc -Wall -O3 -openmp -openmp-report2 matmult.c -o matmult
```

- GCC:

```
module load GnuCC/7.2.0
gcc -Wall -O3 -fopenmp matmult.c -o matmult
```

- Execution:

```
export OMP_DYNAMIC=FALSE
export OMP_NUM_THREADS=16
export GOMP_CPU_AFFINITY="256-271"
./matmult
```

Parallel Loops

```
#pragma omp parallel for private(j,k)
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
       a[i,j] += b[i,k]*c[k,j];
    }
  }
}
```

- Iterations of i-loop are executed by parallel threads.
- Matrix a is shared by all threads.
- Every thread maintains private instances of i, j, k.

Most important source of scalable parallelism.

Load Balancing

```
#pragma omp parallel for ... schedule(kind [, chunk size])
```

- Various kinds of loop scheduling:

static
: Loop is divided into equally sized chunks which are interleaved among threads; default chunk size is N/T.
  - Number of loop iterations N and number of threads T.

dynamic
: Threads retrieve chunks from a shared work queue; default chunk size is 1.

guided
: Like “dynamic” but chunk size starts large and is continuously decremented to specified minimum (default 1).

auto
: One of the above policies is automatically selected (same as if no schedule is given).

runtime
: Schedule taken from environment variable `OMP_SCHEDULE`.

```
export OMP_SCHEDULE="static,1"
```

Example: Matrix Multiplication

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

#define N 2000
double A[N][N], B[N][N], C[N][N];

int main(int argc, char *argv[]) {
  int i, j, k;
  double s;

  for (i=0; i<N; i++)
  {
    for (j=0; j<N; j++)
    {
      A[i][j] = rand();
      B[i][j] = rand();
    }
  }

  printf("%f %f\n", A[0][0], B[0][0]);
  double t1 = omp_get_wtime();
  #pragma omp parallel for private(j,k,s) schedule(runtime)
  for (i=0; i<N; i++)
  {
    for (j=0; j<N; j++)
    {
      s = 0;
      for (k=0; k<N; k++)
      {
        s += A[i][k]*B[k][j];
      }
      C[i][j] = s;
    }
  }

  double t2 = omp_get_wtime();
  printf("%f (%f s)\n", C[0][0], t2-t1);
  return 0;
}
```

Parallel Sections

```
int found1, found2, found3;

#pragma omp parallel sections
{
   #pragma omp section
   found1 = search1();
   #pragma omp section
   found2 = search2();
   #pragma omp section
   found3 = search3();
}

if (found1) printf(“found by method 1\n”);
if (found2) printf(“found by method 2\n”);
if (found3) printf(“found by method 3\n”);
```

- Each code section is executed by a thread in parallel.

Parallel sections and loops may be also nested.

Parallel Blocks

```
int n, a[n], t, i;

#pragma omp parallel private(t, i)
{
  t = omp_get_num_threads(); // number of threads
  i = omp_get_thread_num();  // 0 <= i < t
  compute(a, i*(n/t), min(n, (i+1)*(n/t)));
}
```

- Every thread executes the annotated block.
- Array a and length n are shared by all threads.
- Every thread maintains private instances of t and i.

Parallelism on the lowest level.

Critical Sections

```
int n, a[n], t = 0, i;

#pragma omp parallel private(i)
{
  #pragma omp critical(mutex_i)
  {
    i = t; t++;
  }
  if (i < n) compute(a, i);
}
```

- No two threads can simultaneously execute a critical section with the same name.

High-level but restricted synchronization.

Example: Manual Task Scheduling

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

#define N 2000
double A[N][N], B[N][N], C[N][N];

int main(int argc, char *argv[])
{
  int i, j, k, row;
  double s;

  for (i=0; i<N; i++)
  {
    for (j=0; j<N; j++)
    {
      A[i][j] = rand();
      B[i][j] = rand();
    }
  }

  printf("%f %f\n", A[0][0], B[0][0]);
  double t1 = omp_get_wtime();
  row = 0;
  #pragma omp parallel private(i,j,k,s)
  {
    while (1)
    {
      #pragma omp critical(getrow)
      {
        i = row;
        row++;
      }
      if (i>=N) break;
      for (j=0; j<N; j++)
      {
        s = 0;
        for (k=0; k<N; k++)
        {
          s += A[i][k]*B[k][j];
        }
        C[i][j] = s;
      }
    }
  }
  double t2 = omp_get_wtime();
  printf("%f (%f s)\n", C[0][0], t2-t1);
  return 0;
}
```

Example: Recursive Tasks

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

#define N 3000
double A[N][N], B[N][N], C[N][N];

int matmultrec(int begin, int end);
int matmultrow(int i);

int main(int argc, char *argv[])
{
  int i, j, r;
  double t1, t2;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] = rand();
      B[i][j] = rand();
    }
  }
  printf("%f %f\n", A[0][0], B[0][0]);
  t1 = omp_get_wtime();

  #pragma omp parallel
  {
   #pragma omp single
   {
     r = matmultrec(0, N);
   }
  }

  t2 = omp_get_wtime();
  printf("%d %f (%f s)\n",
    r, C[0][0], t2-t1);
  return 0;
}
```

Example: Recursive Tasks

```
int matmultrec(int begin, int end)
{
  int n = end-begin;
  if (n < 0) return 0;
  if (n == 1)
    return matmultrow(begin);
  int mid = (begin+end)/2;
  int r1, r2;
  #pragma omp task shared(r1)
  r1 = matmultrec(begin, mid);
  #pragma omp task shared(r2)
  r2 = matmultrec(mid, end);
  #pragma omp taskwait
  return r1+r2;
}
```

```
int matmultrow(int i)
{
  int j, k;
  double s;
  for (j = 0; j < N; j++)
  {
    s = 0;
    for (k = 0; k < N; k++)
    {
      s += A[i][k]*B[k][j];
    }
    C[i][j] = s;
  }
  return 1;
}
```

- Recursively create two tasks and wait for their completion.

Task parallelism possible, but may become cumbersome.

1. HPC Architectures

2. OpenMP

3. MPI

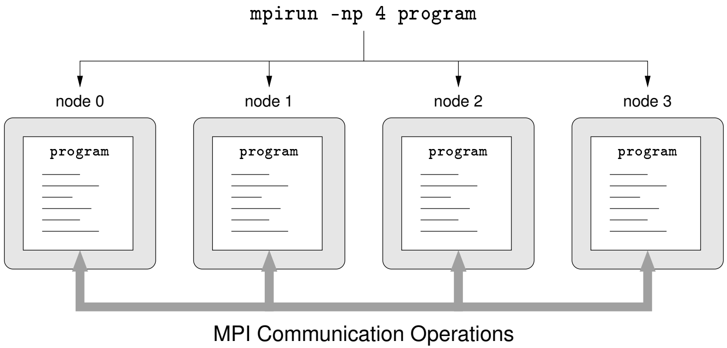
4. Parallel Program Design

Message Passing Interface (MPI)

- An API for portable distributed memory programming.
  - Set of library routines, no compiler support needed.
- Official bindings for C and Fortran.
  - Inofficial bindings exist for various other languages.
- Various implementations.
  - MPICH: initial implementation by Argonne National Lab.
  - MVAPICH: MPICH derivative by Ohio State university.
  - Open MPI: merger of various previous implementations.
  - Commercial implementations by HP, Intel, Microsoft.
  - Hardware support: SGI MPT with MPI offload engine.
- Maintained by the MPI Forum.
  - Current Version: MPI 4.0 (June 2021).

See `http://mpi-forum.org/` for the official specification.

MPI Execution Model



SPMD: Single Program, Multiple Data.

Compiling and Executing MPI

- Paths (Default):

```
CPATH=...:/opt/sgi/mpt/mpt-2.04/include
LIBRARY_PATH=...:/opt/sgi/mpt/mpt-2.04/lib
LD_LIBRARY_PATH=...:/opt/sgi/mpt/mpt-2.04/lib
```

- Source:

```
#include <mpi.h>
```

- Intel Compiler:

```
module load intelcompiler/composer_xe_2015.1.133
icc -std=c99 -Wall -O3 -lmpi matmult.c -o matmult
```

- GCC:

```
module load GnuCC/7.2.0
gcc -Wall -O3 -lmpi matmult.c -o matmult
```

- Execution:

```
export MPI_DSM_CPULIST=32-47
mpirun -np 16 ./matmult 2048
```

A Sample MPI Program

```
#include <mpi.h>
int main(int argc, char *argv[]) {
  char message[20];
  int size, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("processor %d among %d processors\n", rank, size);
  if (rank == 0) {  /* code for process zero */
    strncpy(message,"Hello, there", 19);
    MPI_Send(message, 20, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (rank == 1) { /* code for process one */
    MPI_Status status;
    MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
    printf("received :%s:\n", message);
  }
  MPI_Finalize();
  return 0;
}
```

Basic Operations

```
int MPI_Init(int *argc, char ***argv)

int MPI_Comm_size(MPI_Comm comm, int *size)
IN  comm  communicator (handle)
OUT size  number of processes in the group of comm (integer)

int MPI_Comm_rank(MPI_Comm comm, int *rank)
IN  comm  communicator (handle)
OUT rank  rank of the calling process in group of comm (integer)

int MPI_Finalize(void)

int MPI_Abort(MPI_Comm comm, int errorcode)
IN comm       communicator of tasks to abort
IN errorcode  error code to return to invoking environment
```

Starting a computation, determining its scope, terminating it.

Blocking Send

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype,
  int dest, int tag, MPI_Comm comm)

IN buf       initial address of send buffer (choice)
IN count     number of elements in send buffer (non-negative integer)
IN datatype  datatype of each send buffer element (handle)
IN dest      rank of destination (integer)
IN tag       message tag (integer)
IN comm      communicator (handle)
```

Returns when message buffer may be used again; may (but need not) block, if no matching receive statement was issued.

Blocking Receive

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm, MPI_Status *status)

OUT buf       initial address of receive buffer (choice)
IN count      number of elements in receive buffer (non-negative integer)
IN datatype   datatype of each receive buffer element (handle)
IN source     rank of source or MPI_ANY_SOURCE (integer)
IN tag        message tag or MPI_ANY_TAG (integer)
IN comm       communicator (handle)
OUT status    status object (Status)
```

Blocks until a matching message could be received; if more than one message matches, the first one sent is received.

Example: Computing Pi by Throwing Darts

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

double dboard (int darts);
#define DARTS 50000 /* number of throws */
#define ROUNDS 100  /* number of iterations */
#define MASTER 0    /* task ID of master task */

int main (int argc, char *argv[]) {
  double homepi, pi, avepi, pirecv, pisum;
  int taskid, numtasks, source, mtype, i, n;
  MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
printf ("MPI task %d has started...\n", taskid);

srandom (taskid);
avepi = 0;
for (i = 0; i < ROUNDS; i++) {
   homepi = dboard(DARTS);
   if (taskid != MASTER) {
      mtype = i;
      MPI_Send(&homepi, 1, MPI_DOUBLE,
         MASTER, mtype, MPI_COMM_WORLD);
```

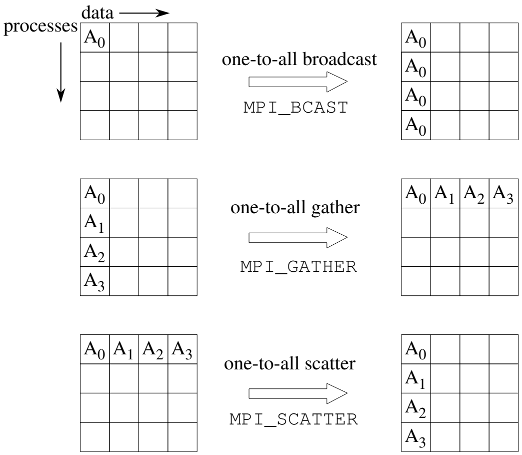
```
  else {
      /* Master receives messages from all workers */
      /* - Message type set to the iteration count */
      /* - Message source set to wildcard DONTCARE: */
      mtype = i;
      pisum = 0;
      for (n = 1; n < numtasks; n++) {
         MPI_Recv(&pirecv, 1, MPI_DOUBLE,
           MPI_ANY_SOURCE, mtype, MPI_COMM_WORLD,
           &status);
         /* keep running total of pi */
         pisum = pisum + pirecv;
         }
      /* Average value of pi for this iteration */
      pi = (pisum + homepi)/numtasks;
      /* Average value of pi over all iterations */
      avepi = ((avepi * i) + pi)/(i + 1);
      printf(" After %8d throws, pi = %10.8f\n",
              (DARTS * (i + 1)),avepi);
      }
   }

if (taskid == MASTER)
   printf ("\nReal value of PI: 3.1415926535897 \n");
MPI_Finalize();
return 0;
}
```

Example: Computing Pi by Throwing Darts

```
#define sqr(x) ((x)*(x))

double dboard(int darts) {

   /* number of hits */
   int score = 0;

   /* throw darts at board */
   for (int n = 1; n <= darts; n++)  {

      /* random coordinates in interval [-1,+1] */
      double r = (double)random()/RAND_MAX;
      double x_coord = (2.0 * r) - 1.0;
      r = (double)random()/RAND_MAX;
      double y_coord = (2.0 * r) - 1.0;

      /* if dart lands in circle, increment score */
      if ((sqr(x_coord) + sqr(y_coord)) <= 1.0)
         score++;
   }

   /* calculate pi = 4*(pi*r^2)/(4*r^2) */
   double pi = 4.0 * (double)score/(double)darts;
   return(pi);
}
```

From MPI tutorial at https://computing.llnl.gov/tutorials/mpi.

Collective Communication



More compact and more efficient programs by the use of collective communication operations.

Broadcast

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
  int root, MPI_Comm comm)

INOUT buffer    starting address of buffer (choice)
IN    count     number of entries in buffer (non-negative integer)
IN    datatype  data type of buffer (handle)
IN    root      rank of broadcast root (integer)
IN    comm      communicator (handle)
```

Sender (root) and receivers perform the same operation.

Gather

```
int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
  void* recvbuf, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)
```

```
IN  sendbuf    starting address of send buffer (choice)
IN  sendcount  number of elements in send buffer (non-negative integer)
IN  sendtype   data type of send buffer elements (handle)
OUT recvbuf    address of receive buffer (choice, significant only at root)
IN  recvcount  number of elements for any single receive
               (non-negative integer, significant only at root)
IN  recvtype   data type of recv buffer elements
               (significant only at root) (handle)
IN  root       rank of receiving process (integer)
IN comm        communicator (handle)
```

Receiver (root) and senders perform the same operation; also the root is one of the senders.

Gather-to-all

```
int MPI_Allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
  void* recvbuf, int recvcount, MPI_Datatype recvtype,
  MPI_Comm comm)
```

```
IN  sendbuf    starting address of send buffer (choice)
IN  sendcount  number of elements in send buffer (non-negative integer)
IN  sendtype   data type of send buffer elements (handle)
OUT recvbuf    address of receive buffer (choice)
IN  recvcount  number of elements received from any process
               (non-negative integer)
IN  recvtype   data type of receive buffer elements (handle)
IN  comm       communicator (handle)
```

Every process serves both as a sender and a receiver.

Scatter

```
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
  void* recvbuf, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)
```

```
IN  sendbuf    address of send buffer (choice, significant only at root)
IN  sendcount  number of elements sent to each process
               (non-negative integer, significant only at root)
IN  sendtype   data type of send buffer elements
               (significant only at root) (handle)
OUT recvbuf    address of receive buffer (choice)
IN  recvcount  number of elements in receive buffer (non-negative integer)
IN  recvtype   data type of receive buffer elements (handle)
IN  root       rank of sending process (integer)
IN  comm       communicator (handle)
```

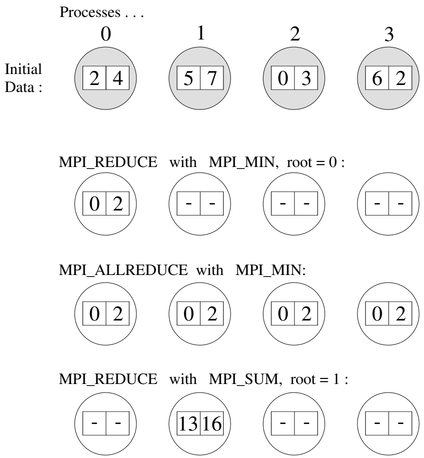
Sender (root) and receivers perform the same operation; also the root is one of the receivers.

Example: Matrix Multiplication

```
int main(int argc, char* argv[]) {
  MPI_Comm comm = MPI_COMM_WORLD;
  int size, rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);

  int n;
  if (rank == 0) {
    if (argc != 2) MPI_Abort(comm, -1);
    n = atoi(argv[1]);
    if (n == 0) MPI_Abort(comm, -1);
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, comm);

  // row number n of A is extended to size*n0
  int n0 = n%size == 0 ? n/size : 1+n/size;
  double *A;
  if (rank == 0) {
    A = malloc(size*n0*n*sizeof(double));
    for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
        A[i*n+j] = rand()%10;
  }
  double* A0 = malloc(n0*n*sizeof(double));
  MPI_Scatter(A, n0*n, MPI_DOUBLE,
    A0, n0*n, MPI_DOUBLE, 0, comm);
```

```
  double* B = malloc(n*n*sizeof(double));
  if (rank == 0) {
    for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
        B[i*n+j] = rand()%10;
  }
  MPI_Bcast(B, n*n, MPI_DOUBLE, 0, comm);

  double* C0 = malloc(n0*n*sizeof(double));
  for (int i=0; i<n0; i++) {
    for (int j=0; j<n; j++) {
      C0[i*n+j] = 0;
      for (int k=0; k<n; k++)
        C0[i*n+j] += A0[i*n+k]*B[k*n+j];
    }
  }

  double* C;
  if (rank == 0)
    C = malloc(size*n0*n*sizeof(double));
  MPI_Gather(C0, n0*n, MPI_DOUBLE,
    C, n0*n, MPI_DOUBLE, 0, comm);
  if (rank == 0) { print(C, n, n); }
  MPI_Finalize();
}
```

Reduction Operations



Processes . . .
0
1
2
3
Initial Data :
2 4
5 7
0 3
6 2
MPI_REDUCE with MPI_MIN, root = 0 :
0 2
- -
- -
- -
MPI_ALLREDUCE with MPI_MIN:
0 2
0 2
0 2
0 2
MPI_REDUCE with MPI_SUM, root = 1 :
- -
13 16
- -
- -

More compact and more efficient programs by the use of reduction operations.

Reduce

```
int MPI_Reduce(const void* sendbuf,
  void* recvbuf, int count, MPI_Datatype datatype,
  MPI_Op op, int root, MPI_Comm comm)

IN  sendbuf   address of send buffer (choice)
OUT recvbuf   address of receive buffer (choice, significant only at root)
IN  count     number of elements in send buffer (non-negative integer)
IN  datatype  data type of elements of send buffer (handle)
IN  op        reduce operation (handle)
IN  root      rank of root process (integer)
IN  comm      communicator (handle)
```

Receiver (root) and senders perform the same operation; also the root is one of the senders.

All-Reduce

```
int MPI_Allreduce(const void* sendbuf,
  void* recvbuf, int count, MPI_Datatype datatype,
  MPI_Op op, MPI_Comm comm)
```

```
IN  sendbuf   address of send buffer (choice)
OUT recvbuf   address of receive buffer (choice)
IN  count     number of elements in send buffer (non-negative integer)
IN  datatype  data type of elements of send buffer (handle)
IN  op        reduce operation (handle)
IN  comm      communicator (handle)
```

Every process serves both as a sender and a receiver.

Predefined Reduction Operations

`MPI_MAX`	maximum
`MPI_MIN`	minimum
`MPI_SUM`	sum
`MPI_PROD`	product
`MPI_LAND`	logical and
`MPI_BAND`	bit-wise and
`MPI_LOR`	logical or
`MPI_BOR`	bit-wise or
`MPI_LXOR`	logical exclusive or (xor)
`MPI_BXOR`	bit-wise exclusive or (xor)
`MPI_MAXLOC`	max value and location
`MPI_MINLOC`	min value and location

Commutative and associative operations; thus the order of reduction does not matter.

User-Defined Reduction Operations

```
int MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)

IN  user_fn  user defined function (function)
IN  commute  true if commutative; false otherwise.
OUT op       operation (handle)
```

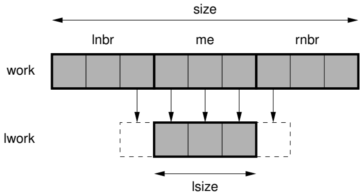
Turn user-defined function to a reduction operation; must be associative but not necessarily commutative.

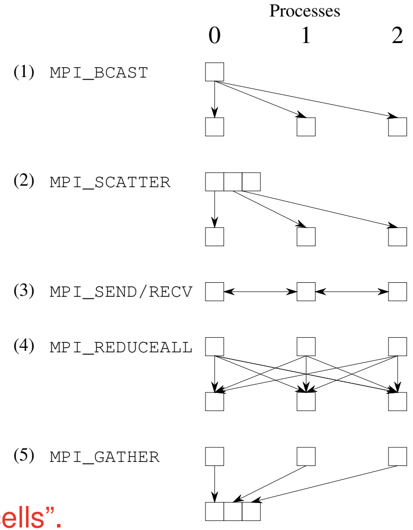
Example: Computing Pi by Throwing Darts

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

double dboard (int darts);
#define DARTS 50000 /* number of throws */
#define ROUNDS 100  /* number of iterations */
#define MASTER 0    /* task ID of master task */

int main (int argc, char *argv[])
{
  double homepi, pisum, pi, avepi;
  int taskid, numtasks, i;
  MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
printf ("MPI task %d has started...\n", taskid);

/* Set seed for random number generator */
srandom (taskid);
```

```
avepi = 0;
for (i = 0; i < ROUNDS; i++) {
   /* All tasks calculate pi */
   homepi = dboard(DARTS);

   /* sum values of homepi across all tasks */
   MPI_Reduce(&homepi, &pisum, 1, MPI_DOUBLE,
     MPI_SUM, MASTER, MPI_COMM_WORLD);

   if (taskid == MASTER) {
      pi = pisum/numtasks;
      avepi = ((avepi * i) + pi)/(i + 1);
      printf("   After %8d throws, pi = %10.8f\n",
              (DARTS * (i + 1)),avepi);
   }
}
if (taskid == MASTER)
   printf ("\nReal PI: 3.1415926535897 \n");
MPI_Finalize();
return 0;
}
```

Example: Finite Difference Problem





Handle boundaries by “ghost cells”.

Example: Finite Difference Problem

```
int main(int argc, char *argv[]) {
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_WORLD, &np);
  MPI_Comm_rank(MPI_WORLD, &me);

  // read work size and work at process 0
  int size; float* work;
  if (me == 0) {
    size = read_work_size();
    work = malloc(size*sizeof(float));
    read_array(work, size);
  }

  // distribute work size to every process
  MPI_Bcast(&size, 1, MPI_INT, 0, MPI_WORLD);

  // allocate space for local work in every process
  if (size%np != 0) MPI_Abort(MPI_WORLD, -1);
  int lsize = size/np;
  float* lwork = malloc((lsize+2)*sizeof(float));

  // distribute work to all processes
  MPI_Scatter(work, lsize, MPI_FLOAT,
    lwork+1, lsize, MPI_FLOAT, 0, MPI_WORLD);

  // determine neighbors in ring
  int lnbr = (me+np-1)%np;
  int rnbr = (me+1)%np;
```

```
  // iterate until convergence
  float globalerr = 99999.0;
  while (globalerr > 0.1) {
    // exchange boundary values with neighbors
    MPI_Send(lwork+1, 1, MPI_FLOAT,
      lnbr, 10, MPI_WORLD);
    MPI_Recv(lwork+lsize+1, 1, MPI_FLOAT,
      rnbr, 10, MPI_WORLD, &status);
    MPI_Send(lwork+lsize, 1, MPI_FLOAT,
      rnbr, 20, MPI_WORLD);
    MPI_Recv(lwork, 1, MPI_FLOAT,
      lnbr, 20, MPI_WORLD, &status);

    // perform local work
    compute(lwork, lsize);

    // determine maximum error among all processes
    float localerr = error(lwork, lsize);
    MPI_Allreduce(&localerr, &globalerr,
      1, MPI_FLOAT, MPI_MAX, MPI_WORLD);
  }

  // collect results at process 0
  MPI_Gather(local+1, lsize, MPI_FLOAT,
    work, lsize, MPI_Float, 0, MPI_WORLD);
  if (me == 0) { write_array(work, size); }
  MPI_Finalize();
}
```

1. HPC Architectures

2. OpenMP

3. MPI

4. Parallel Program Design

Designing Parallel Programs

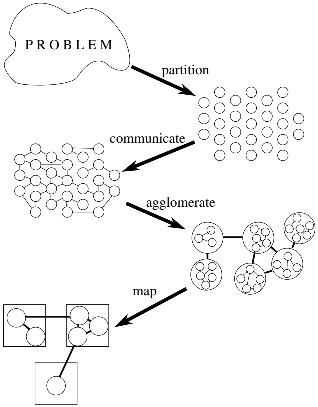
Ian Foster: “Designing and Building Parallel Programs”.

- First consider machine-independent (algorithmic) issues.
  - Concurrency.
  - Scalability.
- Later deal with machine-specific (performance) aspects.
  - Locality.
  - Placement.

A methodological approach in multiple stages.

The PCAM Approach

- Partitioning.
  - Decompose computation and data.
  - Exhibit opportunities for parallelism by creating many small tasks.
- Communication.
  - Analyze data dependencies.
  - Determine structure of commmunication and coordination.
- Agglomeration.
  - Combine tasks to bigger tasks.
  - Improve performance of execution on real computers.
- Mapping.
  - Assign tasks to processors.
  - Maximize utilization and minimize communication.



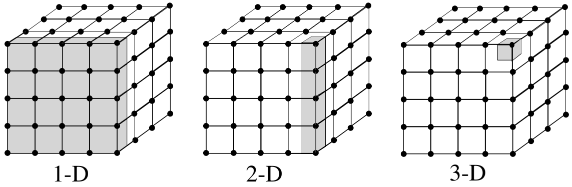
Partitioning

Expose opportunities for parallelism.

- Construct fine-grained decomposition of problem.
  - Domain/data decomposition:
    - Partition data, associate computation to data.
  - Functional/task decomposition:
    - Partition computation, associate data to computation.
- Complementary approaches.
  - Should be both considered.
  - Can lead to alternative algorithms.
  - Can be applied to different parts of problem.
- Avoid replication of computation or data.
  - May be introduced later to reduce communication overhead and to increase the granularity of tasks.

Domain Decomposition

Focus on the decomposition of the data.

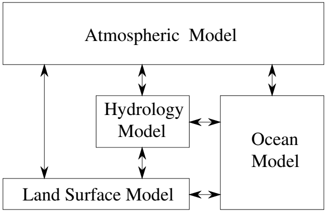


- Divide data into small pieces and associate computation.
  - If computation requires several, associate to “main” piece.
  - Communication is required for access to the other pieces.
- Resulting tasks should be of roughly the same size.
  - Otherwise load balancing may become difficult.
- Prefer finer decomposition over coarse ones.
  - Small tasks may be agglomerated in later stage.

Typical for problems with large central data structures.

Functional Decomposition

Focus on the decomposition of the computation.



- Decompose according to the algorithmic structure.
  - Independent computational blocks.
  - Independent loop iterations.
  - Independent (recursive) function invocations.
- Determine data requirements of each task.
  - If requirements overlap, communication is required.

Typical for problems without central data structures.

Partitioning Design Checklist

- Is number of tasks large enough?
  - Order of magnitude larger than processor number.
  - Keeps flexibility for further stages.
- Does number of tasks scale with problem size?
  - Larger problems can be solved with more processors.
- Are the tasks of comparable size?
  - Otherwise load balancing may become difficult.
- Are redundant computations and data avoided?
  - Otherwise scalability may suffer.
- Have alternative partitions been considered?
  - Try both domain and functional decomposition.

Do we have sufficient concurrency?

Communication

Specify flow of information between tasks.

- Describe communication structure by “channels”.
  - Connections between those tasks that produce data and those that consume them.
  - Typically easy to determine for functional decomposition from data flow between tasks.
  - May be complex to determine for domain decomposition due to data dependencies.
- Analyze the usage of channels.
  - Number and sizes of messages flowing through channels.
  - Temporal relationship/dependencies between messages flowing through different channels.

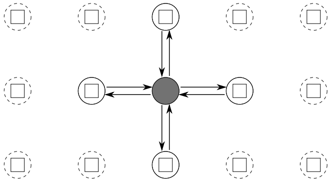
Also a healthy exercise for shared memory programs.

Types of Communication

- Local versus global:
  - Communication with a small set of tasks (“neighbors”) or with many other tasks.
- Structured versus unstructured:
  - Communication forms a regular structure (tree, grid, . . .) or an arbitrary graph.
- Static versus dynamic:
  - Identity of communication partners is known in advance and does not change or depends on runtime data and may vary.
- Synchronous versus asynchronous:
  - Producers and consumers cooperate in data transfer or consumer may acquire data without producer cooperation.

Local Communication

Example: Jacobi finite differences method.



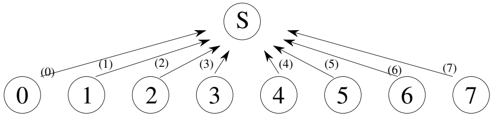
$$X_{i,j}^{t+1} = \frac{1}{8}\left(4X_{i,j}^{t} + X_{i-1,j}^{t} + X_{i+1,j}^{t} + X_{i,j-1}^{t} + X_{i,j+1}^{t}\right)$$

```
for t=0 to T-1 do
  send X(i,j) to each neighbor
  receive X(i-1,j), X(i+1,j), X(i,j-1), X(i,j+1) from neighbors
  update X(i,j)
end
```

Global Communication

Example: parallel reduction operation.

$$S = \sum_{i=0}^{n} X_i$$

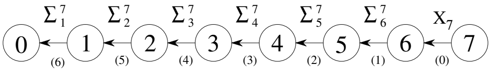


- Centralized algorithm:
  - Single task becomes bottleneck of communication and computation.
- Sequential algorithm:
  - Additions are performed one after each other.

Global Communication

Example: parallel reduction operation.

$$\sum_{i=j}^{n} X_i = X_j + \sum_{i=j+1}^{n} X_i$$

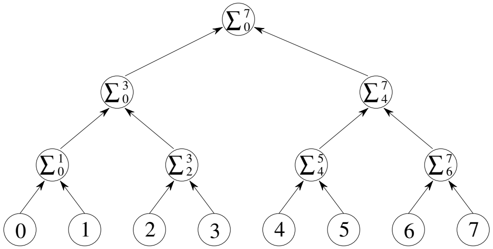


- Decentralized algorithm:
  - Communication/computation are distributed among tasks.
- But still a sequential algorithm.

Global Communication

Example: parallel reduction operation.

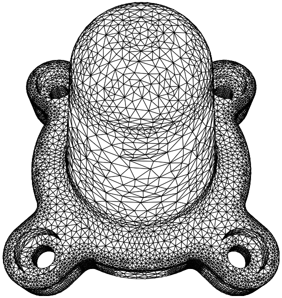
$$\sum_{i=j}^{j+k} X_i = \left(\sum_{i=j}^{j+\lfloor k/2 \rfloor} X_i\right) + \left(\sum_{i=j+\lfloor k/2 \rfloor+1}^{j+k} X_i\right)$$



- Decentralized and parallel algorithm:
    - Up to $k/2$ additions can be performed in parallel.

Unstructured/Dynamic Communication

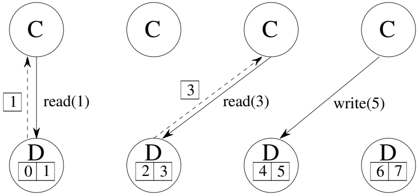
Example: finite element method.



- Mesh of points representing a physical object.
  - Simulation of, e.g., the impact of force on the object.
  - Shape of the mesh is modified by the impact.
- Domain decomposition.
  - Unstructured communication: mesh is irregular.
  - Dynamic communication: mesh changes.

Asynchronous Communication

Example: management of a shared data structure.



- A set of “data tasks” manages a shared data structure.
    - Data structure is distributed among tasks.
- A set of “computing tasks” produce and consume data.
    - Exchange of messages between computing tasks and data tasks for reading and writing the data structure.

Consumption of data decoupled from their production.

Communication Design Checklist

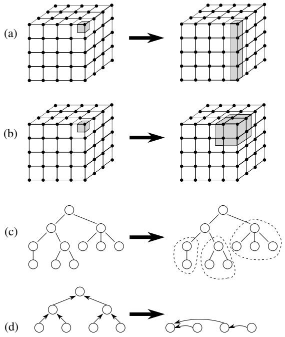
- Do all tasks perform the same amount of communication?
- Does each task communicate only with a few neighbors?
- Can the communication operations proceed concurrently?
- Can the computation operations proceed concurrently?

Do we have the potential for scalability?

Agglomeration

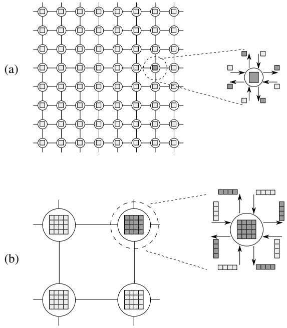
In the previous phases we have developed a parallel algorithm.

- Algorithm not efficiently executable.
    - Large number of small tasks.
    - Large amount of communication.
- Combine tasks to larger tasks.
    - Increase the granularity of tasks.
        - Granularity: the ratio of computation to communication.
    - Still retain design flexibility.
        - Sufficiently many tasks for scalability and mapping flexibility.
    - Reduce engineering costs.
        - Avoid effort of parallelization where it does not pay off.



Increasing Granularity: Surface to Volume

- Before: granularity $1/4 = 0.25$.
  - 1 local computation operation.
  - 4 data items sent.
- After: granularity $16/16 = 1$.
  - 16 local computation operations.
  - 16 data items sent.
- Surface to Volume Effect
  - Typical for domain decomposition.
  - Communication proportional to "surface" of subdomain.
  - Computation proportional to "volume" of subdomain.
  - Surface grows slower than volume.
    - Square: $S/V = 4a/a^2 = 4/a$.

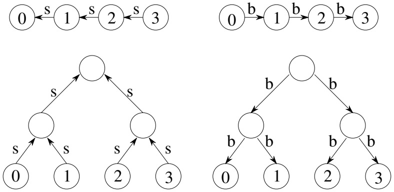


Decreasing surface-to-volume ratio increases granularity.

Increasing Granularity: Replicating Computation

Communication may be decreased by replicating computation.

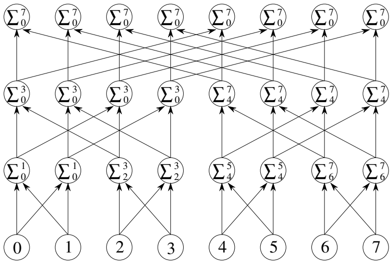
Example: two algorithms computing a global sum in N tasks.



Time $2(N-1)$ resp. $2\log_2 N$ for performing $N-1$ additions.

Increasing Granularity: Replicating Computation

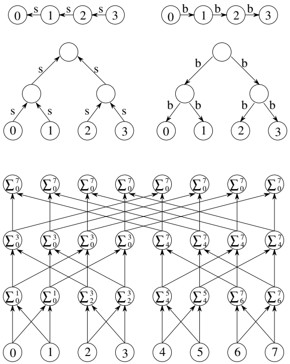
A replicating algorithm computing a global sum in N tasks.

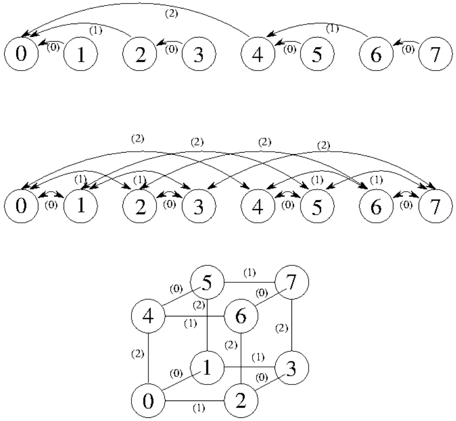


Time $\log_2 N$ for performing $N \log N$ additions.

Increasing Granularity: Avoiding Communication

Agglomerate tasks that cannot execute concurrently.





Only N agglomerated tasks are needed.

Retaining Design Flexibility

Do not “over-agglomerate”.

- Goal is not a fixed number of tasks.
  - Task number should grow with problem and machine size.
  - Algorithm should remain scalable.
- Goal is not one task per processor.
  - There shold be still multiple tasks per processor.
  - If one task is blocked, another one may execute and keep the processor busy.

Agglomeration should not “hardwire” the algorithm to a fixed problem and machine size.

Reducing Engineering Costs

- Try to avoid extensive code changes.
  - One partitioning/agglomeration may be much more difficult to implement than another.
- Try to avoid extensive data structure changes.
  - Conversions from/to data structures given by the context of the parallel application may be cumbersome.

Consider also the costs of development in relation to the expected performance gains.

Agglomeration Design Checklist

- Has communication been reduced (granularity increased)?
- Does computation replication outweigh its costs?
- Does data replication not limit scalability?
- Have tasks still similiar sizes?
- Is there still sufficient concurrency?
- Does the number of tasks still scale with problem size?
- Can task number be reduced without limiting flexibility?
- Are the engineering costs reasonable?

Do we have sufficient execution efficiency?

Mapping

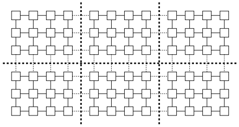
We need a strategy for mapping tasks to processors (cores).

- Only a problem for systems with distributed memory or shared memory with non-uniform memory access.
  - On multi-core processors and SMP systems, the automatic placement of tasks to cores by the OS suffices.
- Conflicting goals:
  - Place tasks that are able to execute concurrently on different processors.
  - Place tasks that communicate frequently on the same processor.

The mapping problem is NP-complete, so we can in general only hope for good heuristics.

Types of Mapping

- Static mappings:
  - A fixed number of permanent tasks is mapped at program start to processors; this mapping does not change.

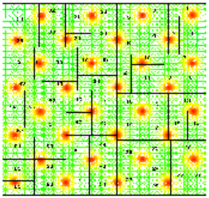


- Load balancing algorithms:
  - The assignment of permanent tasks to processors is adapted at runtime to keep processors equally busy.
- Task scheduling algorithms:
  - Many short-living tasks are created at runtime; a scheduler maps tasks to processors where they run until termination.

Static mapping is usually only sufficent for domain decomposition with structured communication.

Static Mappings: Recursive Bisection

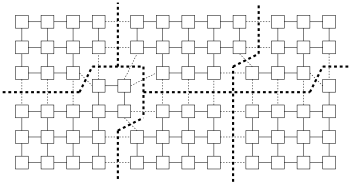
Recursively divide domain into partitions with equal costs.

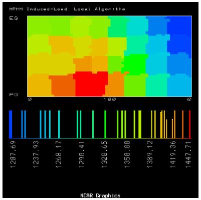


- Recursive coordinate bisection:
  - Recursively cut multi-dimensional grid at longest dimension.
- Unbalanced recursive bisection:
  - Choose among partitions the one with lowest aspect ratio.
- Recursive graph bisection:
  - Decompose graph according to distance from extremities.

Load Balancing: Local Algorithms

Compare load with that neighbor processors; transfer load if difference gets too big.

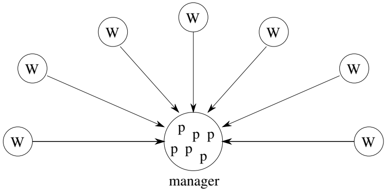




Use only local information and that of neighbor processors.

Task Scheduling

Maintain pool of tasks to which all new tasks are added.



- Manager/worker scheme:
  - Manager controls pool; idle workers ask manager for tasks.
- Hierarchical manager/worker scheme:
  - Subsets of workers with own submanagers and subpools.
  - Submanagers interact with manager (and each other).
- Decentralized schemes:
  - Each worker maintains its own task pool.
  - Idle workers request tasks from other workers.

Termination detection may become an issue.

Mapping Design Checklist

- If considering a program where tasks are only created at startup, have you also considered task scheduling?
- If considering task scheduling, have you also considered a program where tasks are only created at startup?
- If considering load-balancing, have you evaluated simpler alternatives such as probabilistic or cyclic mappings?
- If considering probabilistic or cyclic mappings, have you verified that task number is large enough to balance load?
- If considering task scheduling, have you verified that the manager does not become a bottleneck?

Do we have sufficient processor utilization?

General Recommendations

- Be sure to parallelize the actual hotspots of a program.
  - First you must understand where computation time is spent.
- Consider alternatives.
  - Do not just implement the first scheme that comes to mind.
- Remember scalability.
  - You may get more cores available than originally thought.
- But also consider the coding effort.
  - A simple solution may be sufficient as a starting point.
- And do not forget the application context.
  - The parallel code must be integrated into a bigger system.

Ultimately, determining the most efficient parallelization strategy for a given problem may require multiple iterations of performance debugging and optimizing/rewriting the code.