Building and Running a Parallel Application

Steve Lantz
Computing and Information Science 4205
www.cac.cornell.edu/~slantz

Unix Trix

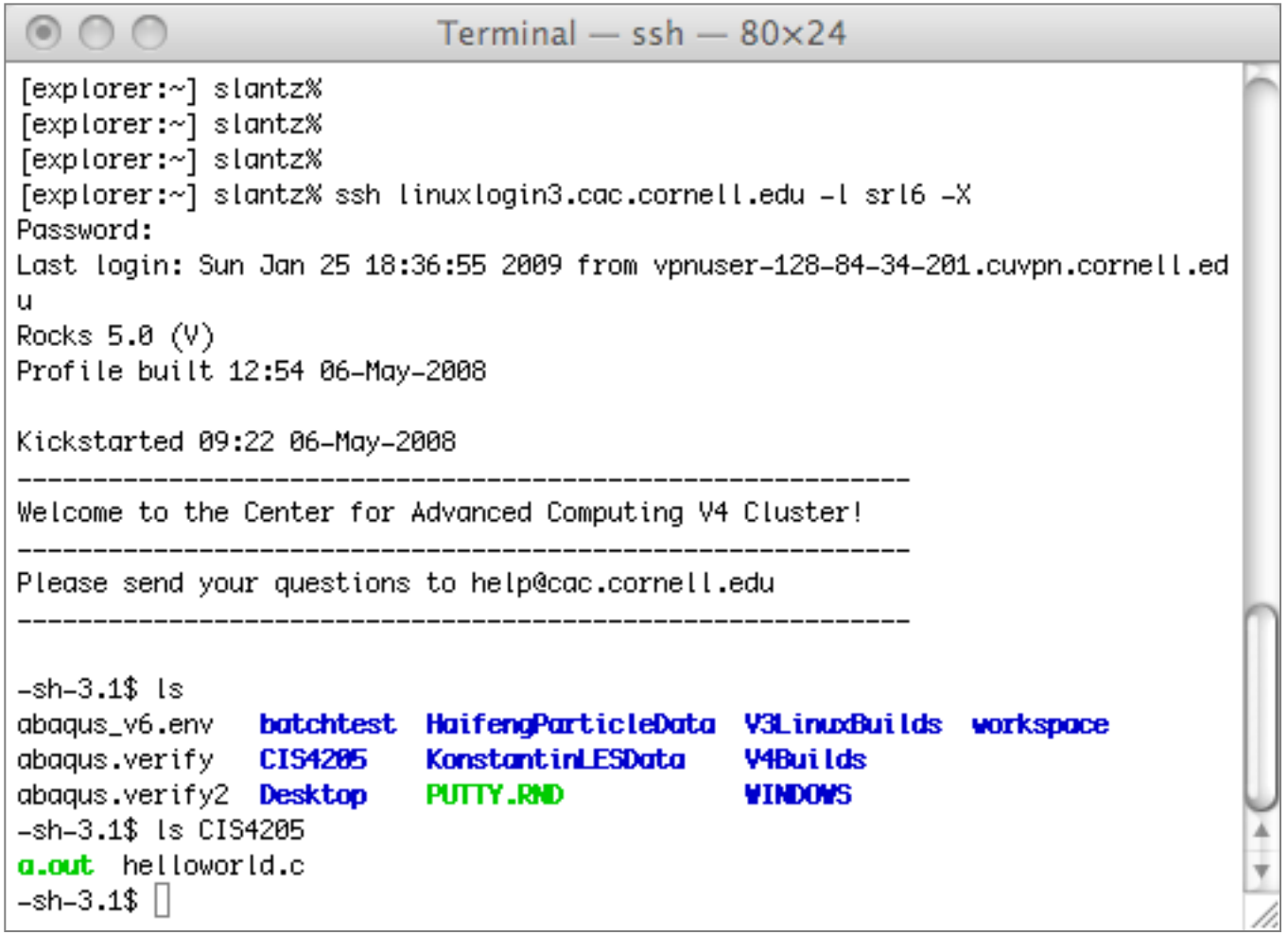
Unix/Linux Commands to Know and Cherish

- **Shell: bash or tcsh**
  - The shell defines many of the commands you enter at the command line
  - The Bourne Again Shell (bash) is an update to the original Bourne shell (sh)
  - Similarly tcsh is an update to csh, the C Shell (up-arrow to get last command)
- **man = “manual” = the way you get help, e.g., “man ls”**
- **Working with directories: cd, pwd, ls, mkdir, rmdir**
  - cd to change directory (popd, pushd to use directory stack); “cd ..” = up a level
  - pwd = print working directory = print your current location (also known as .)
  - “ls -l” gives you complete directory listing, “ls -a” lets you see .prefix-files
  - mkdir to create a new directory, rmdir to remove an existing one
- **Environment variables: export (bash, sh) or setenv (tcsh, csh)**
  - Variables that are local to the shell are defined with “set”
  - Env variables are inherited by shells started in the parent shell
  - Type “set” to see locals, “env” to see environment

Unix/Linux Commands to Know and Cherish, 2

- **To view an environment variable: “echo $varname”**
- **Move, copy, remove files: mv, cp, rm**
- **To view the contents of a file: “cat filename”**
  - cat = “concatenate to standard output”, stdout is the terminal by default
- **Redirect stdout using symbols**
  - “cat file1 > file2” replaces (clobbers) file2 with the contents of file1
  - “cat file1 >> file2” appends file2 with the contents of file1
  - “cmd1 | cmd2” to pipe stdout of cmd1 to stdin of cmd2
- **Text editors: vi, emacs**
  - Terminal window becomes plain text editor
  - No graphical interface, all editing done via special key sequences
- **Controlling processes**
  - control sequences: ctrl-c = kill, ctrl-z = suspend
  - bg to put process in background, fg to bring to foreground, “jobs” to see bg list

Remote Shell and Secure Shell



What Happens When You Run an MPI Program?

- Command *mpiexec* (mpirun in some implementations) launches multiple processes on same or different machines
- The remote processes are launched using *ssh*
- All processes are copies of the same program, yet they are completely independent processes
- mpiexec assigns a unique rank to each process, which becomes part of that process's environment
- A process contacts the local MPI *daemon* (*mpd*) running on each machine in order to communicate with other MPI processes
- The mpd knows about the other processes that are part of the same MPI job, and their ranks

- Demonstration of compiling and running helloworld.c

Testing Your MPI Setup

Start up some daemons

mpdboot -n <numberofhosts>

If mpdboot doesn't work, here is plan B for starting the mpd daemons:

At the command prompt, enter: mpd &
Run mpdtrace -l to find out the port number mpd is running on
To start mpd's on the other machines, run:
ssh <nextmachinename> mpd -h <firstmachinename> -p <port> -d

Once all the daemons are running, see what is going on with MPICH

Run mpdtrace to get a quick trace and hopefully see all the nodes
Run mpdringtest 3000 to run a ring around the mpd daemons
Verify you've got the right hosts with:

mpiexec -n <numberofhosts> hostname

In Memoriam:
Prof. Ravindra Nath Sudan, 1931-2009

- Emigrated to the U.S. in 1958 to join the faculty of Cornell University in electrical engineering
- Eventually became IBM Professor of Engineering
- Director of the Laboratory of Plasma Studies (LPS) from 1975-85
- Co-PI on the original proposal that created the Cornell Theory Center
- Deputy director of CTC, 1985-87
- Winner of the James Clerk Maxwell Prize from the American Physical Society in 1989
- My Ph.D. thesis adviser

A Great Online Linux Tutorial!

This website is beautifully elegant; succinct, yet complete. I am in awe.

I hereby give it its own slide as a tribute:http://www.ee.surrey.ac.uk/Teaching/Unix/index.html

It consists of an intro and 8 tutorial pages. I recommend them all.

Assignment due 1/30:

- **Download the gzipped tar file from Tutorial 7 and build the code within it on any Linux computer by following the steps delineated in 7.2 – 7.6.**
- **Run the code and email me *all* the output you get.**
- **For input, use the numerical part of your Cornell NetID as the number of feet, and ask for output in meters. Example: my NetID is srl6, so my input would be "6 feet".**
- **In the process, you will learn how to run configure and make!**

Another One of My Heroes From the UK

PuTTY is a very nice, free ssh client for Windows. Get it!

http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Gotta love Simon.

Parallel Algorithm Design (Foster's Method)*

* Designing and Building Parallel Programs
Concepts and Tools for Parallel Software Engineering
By: Ian Foster

Task/Channel Model

- **Design Efficient Parallel Programs (or Algorithms)**
  - Mainly for distributed memory systems (e.g. Clusters)
- **Break Parallel Computations into:**
  - Tasks (program solving part of a problem, memory & I/O ports)
  - Channels (message queue from one Tasks output I/O port to another's input I/O port
- **Communication Specifics**
  - Tasks receive data from other Tasks via Channels
    - Receives are synchronous (task blocks until desired message is received)
  - Tasks send data to other Tasks via Channels
    - Sends are asynchronous (messages are sent and work continues)
  - Note: This is how MPI_Send & MPI_Recv api calls work

Foster's Design Methodology

- **Partitioning**
  - Dividing the Problem into Tasks
- **Communication**
  - Determine what needs to be communicated between the Tasks over Channels
- **Agglomeration**
  - Group or Consolidate Tasks to improve efficiency or simplify the programming solution
- **Mapping**
  - Assign tasks to the Computer Processors
    - (assume distributed-memory system e.g. Cluster)

Illustration of the Four Steps

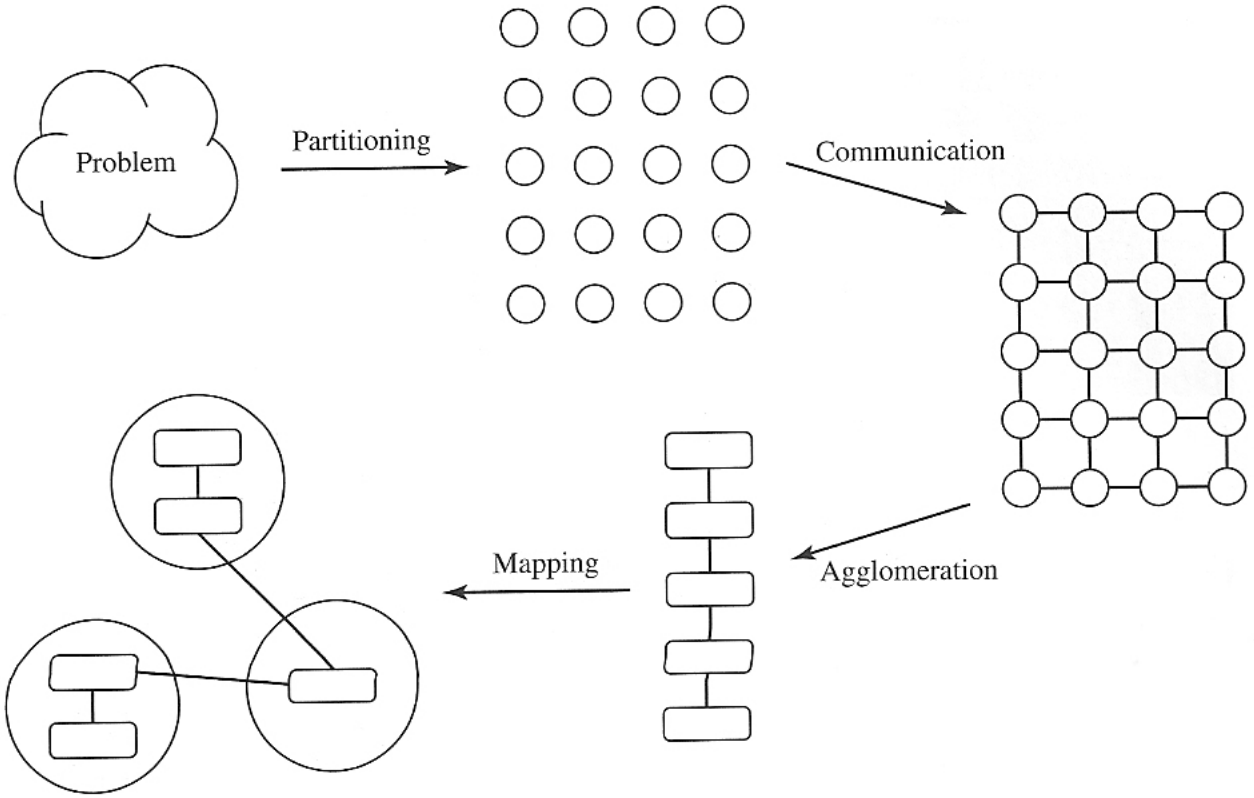


Figure 3.2 Foster's parallel algorithm design methodology.

Step 1: Partitioning
Divide Computation & Data into Pieces

- **Domain Decomposition – Data Centric Approach**
  - Divide up most frequently used data
  - Associate the computations with the divided data

- **Functional Decomposition – Computation Centric Approach**
  - Divide up the computation
  - Associate the data with the divided computations

- **Primitives: Resulting Pieces from either Decomposition**
  - The goal is to have as many Primitives as possible

Domain Decomposition Example

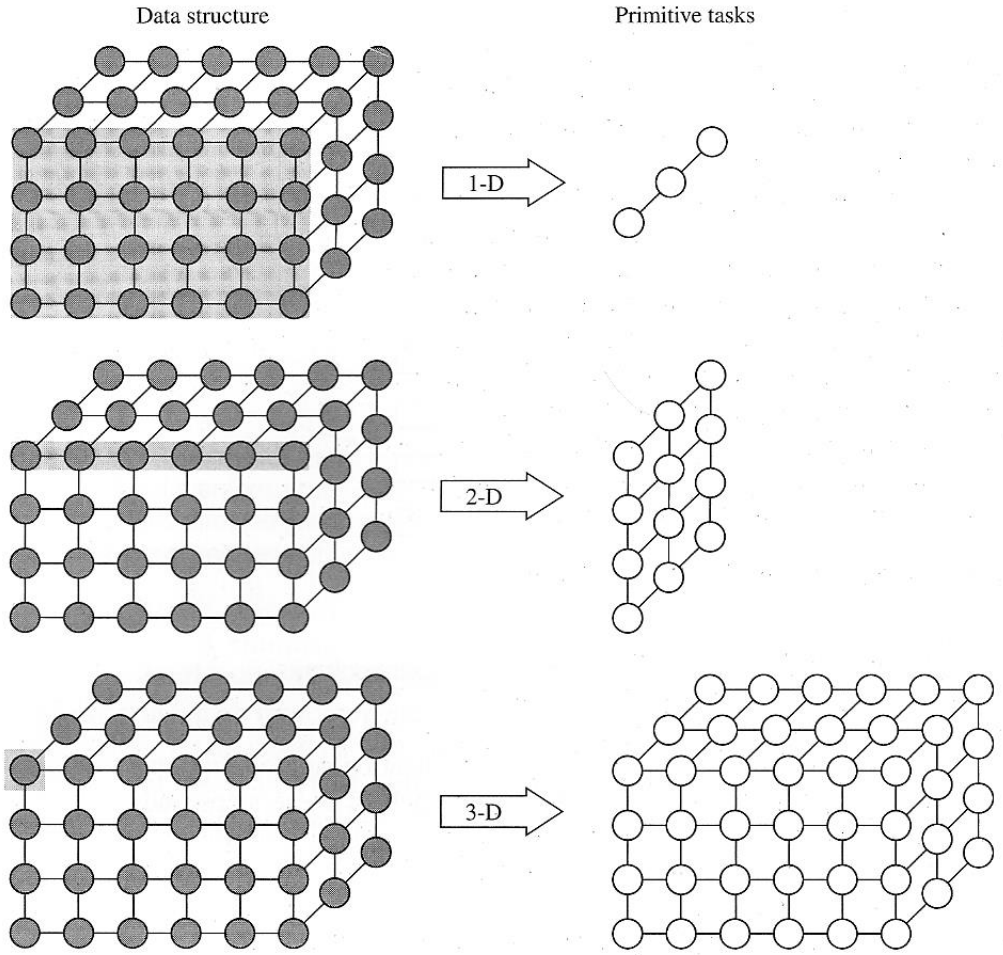


Figure 3.3 Three domain decompositions of a three-dimensional matrix, resulting in markedly different collections of primitive tasks.

Functional Parallelism Example

Could this be pipelined?

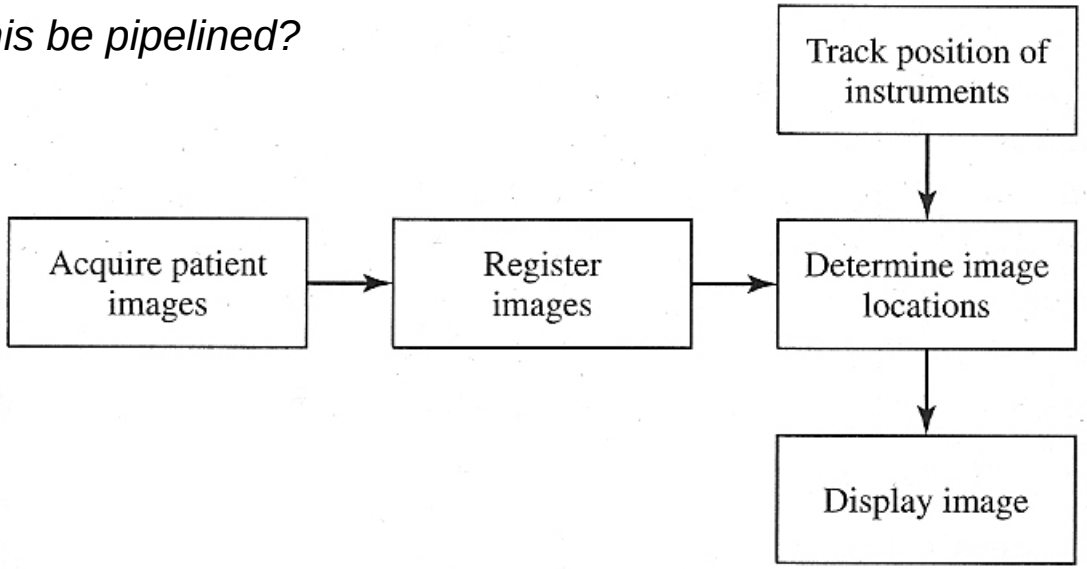


Figure 3.4 Functional decomposition of a system supporting interactive image-guided surgery.

Partitioning Goals

- **Order of magnitude more Primitive tasks than Processors**
- **Minimize redundant computations and data**
- **Primitive tasks are approximately the same size**
- **The number of Primitive tasks increase as problem size increases**

Step 2: Communication
Determine Communication Patterns between Primitive Tasks

- **Local Communication**
  - When Tasks need data from a small number of other Tasks
  - Channel from Producing Task to Consuming Task Created

- **Global Communication**
  - When Task need data from many or all other Tasks
  - Channels for this type of communication are not created during this step

Communication Goals

- **Communication is balanced among all Tasks**
- **Each Task Communicates with a minimal number of neighbors**
- **Tasks can Perform Communications concurrently**
- **Tasks can Perform Computations concurrently**

Note: *Serial codes do not require communication. When adding communication to parallel codes, consider this an overhead because Tasks cannot perform Computations while waiting for data. If not done carefully, the cost of communication can outweigh the performance benefit of parallelism.*

Step 3: Agglomeration
Group Tasks to Improve Efficiency or Simplify Programming

- **Increase Locality**
  - remove communication by agglomerating Tasks that Communicate with one another
  - Combine groups of sending & receiving task
    - Send fewer, larger messages rather than more short messages which incur more message latency.

- **Maintain Scalability of the Parallel Design**
  - Be careful not to agglomerate Tasks so much that moving to a machine with more processors will not be possible

- **Reduce Software Engineering costs**
  - Leveraging existing sequential code can reduce the expense of engineering a parallel algorithm

Illustration of Agglomeration

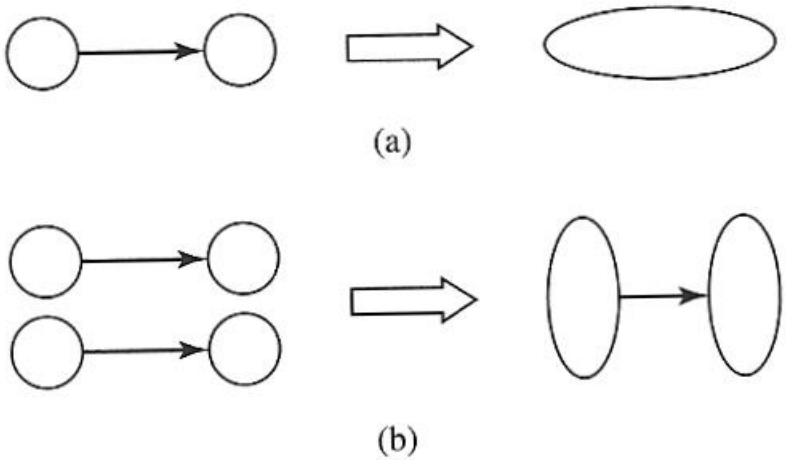


Figure 3.5 Agglomerating tasks can eliminate communications or at least reduce their overhead. (a) Combining tasks that are connected by a channel eliminates that communication, increasing the locality of the parallel algorithm. (b) Combining sending and receiving tasks reduces the number of message transmissions.

Agglomeration Goals

- Increase the locality of the parallel algorithm
- Replicated computations take less time than the communications they replace
- Replicated data is small enough to allow the algorithm to scale
- Agglomerated tasks have similar computational and communications costs
- Number of Tasks can increase as the problem size does
- Number of Tasks as small as possible but at least as large as the number of available processors
- Trade-off between agglomeration and cost of modifications to sequential codes is reasonable

Step 4: Mapping
Assigning Tasks to Processors

- **Maximize Processor Utilization**
  - Ensure computation is evenly balanced across all processors
- **Minimize Interprocess Communication**

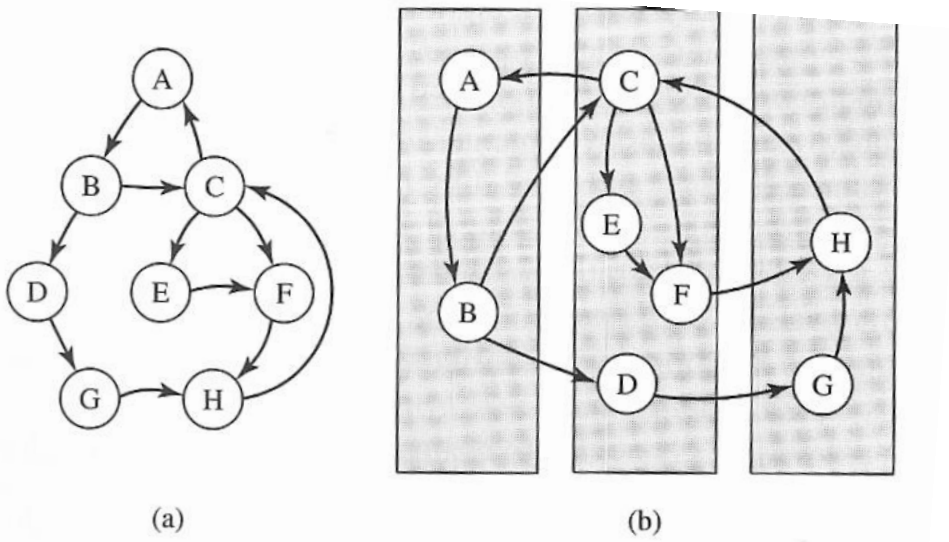


Figure 3.6 The mapping process. (a) A task/channel graph. (b) Mapping of tasks to three processors. Some channels now represent intraprocessor communications, while others represent interprocessor communications.

Mapping Goals

- **Mapping based on one task per processor and multiple tasks per processor have been considered**
- **Both static and dynamic allocation of tasks to processors have been evaluated**
- **If a dynamic allocation of tasks to processors is chosen, the Task allocator is not a bottleneck**
- **If Static allocation of tasks to processors is chosen, the ratio of tasks to processors is at least 10 to 1**

Decision Tree for Parallel Algorithm Design

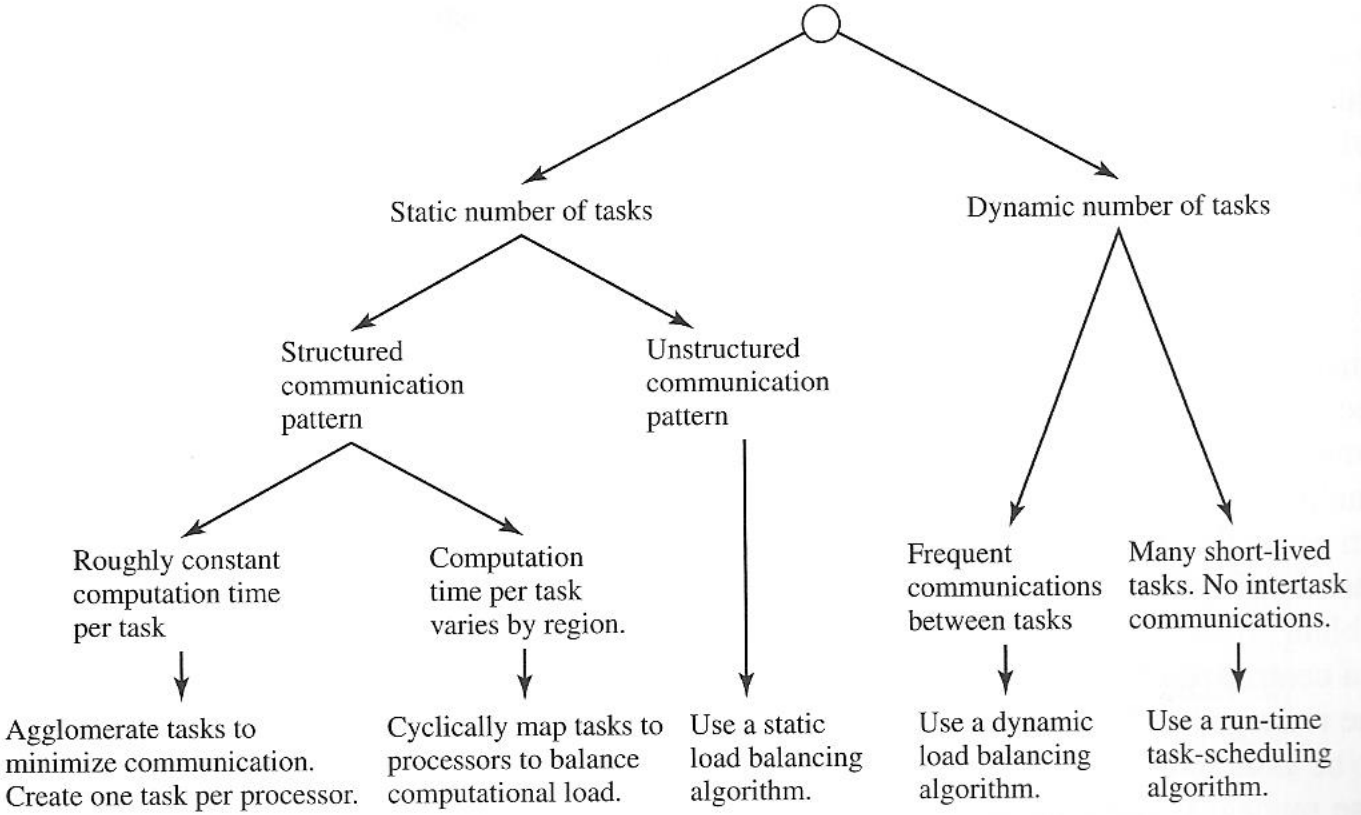


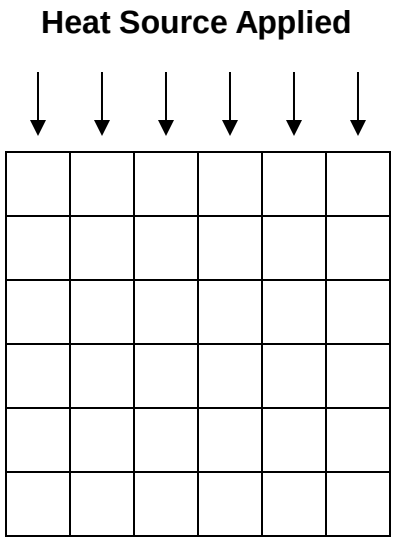
Figure 3.7 A decision tree to choose a mapping strategy. The best strategy depends on characteristics of the tasks produced as a result of the partitioning, communication, and agglomeration steps.

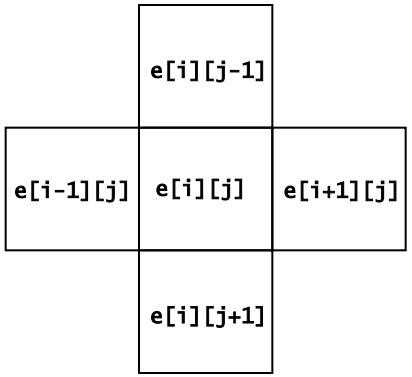
Foster's Method* in Action

* Designing and Building Parallel Programs
Concepts and Tools for Parallel Software Engineering
By: Ian Foster

Heat Transfer Problem
Jacobi Method

- **2 Dimensional Grid Modeling a steel plate**
- **A simulated heat source is applied to one the "top" boundary**
- **Simulation is run over a number of time-steps**
- **For each time step, new values for each element of the grid are calculated until the values converge. (Jacobi Method)**
  - Values for each element are based on the values of its neighbor above, below, left & right





Heat Transfer Problem
Serial C Solution 1 of 4

```
#include <stdio.h>
#include <math.h>
#define EPSILON 0.00001
#define N 100
#define time_steps 100

int main (int argc, char *argv[])
 {
  int i,j;
  int step;
  double time;
  double eps, enew;
  double time_max = 3.0;
  double alpha = 0.06;
  double dx = 1.0/N;
  double dy = 1.0/time_steps;
  double dt = time_max/time_steps;
  double dxinv = 1.0/dx;
  double dyinv = 1.0/dy;
  double dtinv = 1.0/dt;
  double divinv = 1.0/(dtinv + 2 * alpha * (dxinv * dxinv + dyinv * dyinv));
  double t[N][N];
  double told[N][N];
  double minval, maxval;
  long clock(),cputime;
  char fname[40];
  FILE *out;
```

Heat Transfer Problem
Serial C Solution 2 of 4

```
clock();
// initialize interior values
for (i=1; i<(N-1); i++)
 for (j=1; j<(N-1); j++)
  told[i][j] = 0.0;
 // set initial boundary conditions
 for (i=0; i<N; i++)
  {
   told[i][0]   = 0.0; // left
   told[i][N-1] = 0.0; // right
  }
 for (j=0; j<N; j++)
  told[N-1][j] = 0.0; // bottom
// for all time steps
for (step = 1; step <= time_steps; step++)
 {
  time = step * (time_max/time_steps);
  // reset top boundary condition each timestep
  for (j=0; j<N; j++)
   told[0][j]   =  2.0 * sin(time); // top
```

Heat Transfer Problem
Serial C Solution 3 of 4

```
do
 {
  eps = 0.0;
  for (i=1; i<(N-1); i++)
   for (j=1; j<(N-1); j++)
    t[i][j]=((told[i][j+1]+told[i][j-1])*alpha*dyinv*dyinv+
(told[i+1][j]+told[i-1][j])*alpha*dxinv*dxinv+
(told[i][j]*dtinv))*divinv;
  for (i=1; i<(N-1); i++)
   {
    for (j=1; j<(N-1); j++)
     {
      enew = fabs(t[i][j] - told[i][j]);
      if (enew > eps) { eps = enew; }
     }
   }
  for (i=0; i<N; i++)
   for (j=0; j<N; j++)
    told[i][j] = t[i][j];
 }
while(eps > EPSILON);
```

Heat Transfer Problem
Serial C Solution 4 of 4

```
   // Dump raster date to a file
   minval = 0.0;
   maxval = 0.0;
   for (i=0; i<N; i++)
    {
     for (j=0; j<N; j++)
      {
       if (t[i][j] < minval) { minval = t[i][j]; }
       if (t[i][j] > maxval) { maxval = t[i][j]; }
      }
    }
   sprintf(fname,"Output\\heat%03d.raw",step);
   out = fopen(fname,"w+b");
   for (i=0; i<N; i++)
    for (j=0; j<N; j++)
     fprintf(out,"%c",(int)(((t[i][j]-minval)*255.0)/(maxval - minval)));
   fclose(out);
   printf("Time step: %d\r",step);
  } // for all time steps
 cputime = clock();
 printf("%d time steps in %.2f seconds\n",step-1,cputime/1.0e+3);
}
```

Step 1: Partitioning
Divide Computation & Data into Pieces

- **The Primitive task would be Computing each element in the Grid**

Goals:

✓ **Order of magnitude more Primitive tasks than Processors**

✓ **Minimize redundant computations and data**

✓ **Primitive tasks are approximately the same size**

✓ **The number of Primitive tasks increase as problem size increases**

Step 2: Communication
Determine Communication Patterns between Primitive Tasks

- **Each Primitive task needs an input channel to 4 neighbors**
- **Each Primitive task needs an output channel to 4 neighbors**

Goals:

✓ **Communication is balanced among all Tasks**

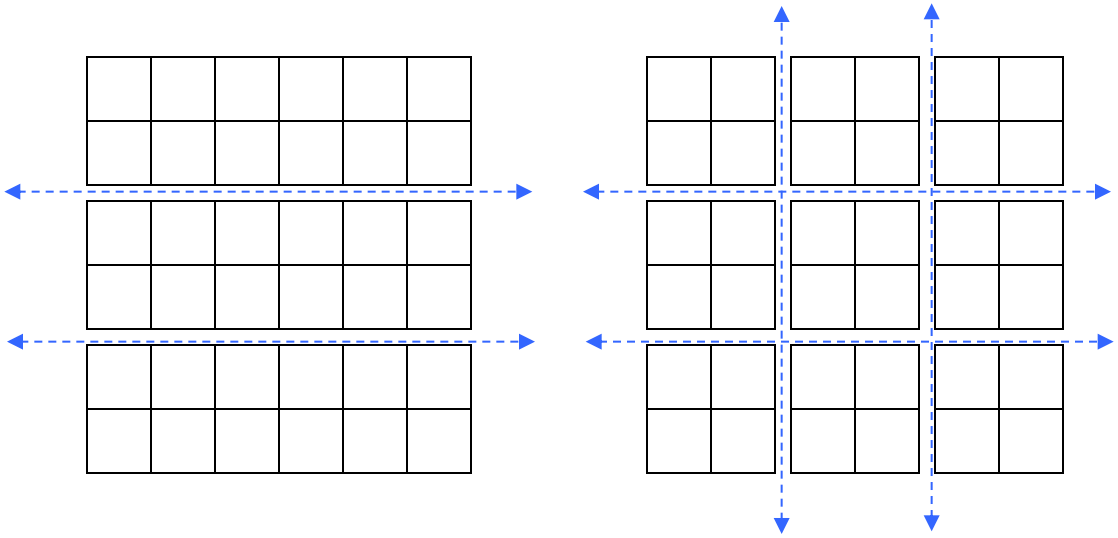
✓ **Each Task Communicates with a minimal number of neighbors**

✓ **Tasks can Perform Communications concurrently**

✓ **Tasks can Perform Computations concurrently**

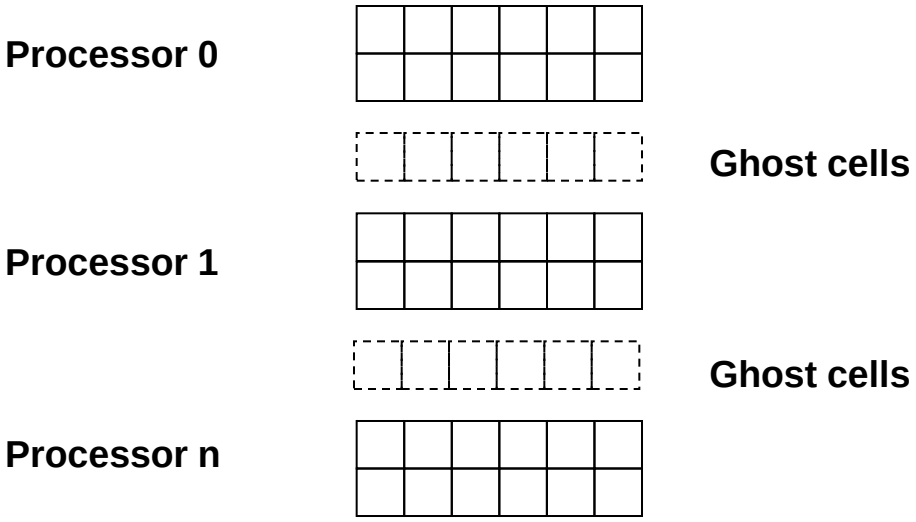
Step 3: Agglomeration

Group Tasks to Improve Efficiency or Simplify Programming



- ✓ Increase the locality of the parallel algorithm
- ✓ Replicated computations take less time than the communications they replace
- ✓ Replicated data is small enough to allow the algorithm to scale
- ✓ Agglomerated tasks have similar computational and communications costs
- ✓ Number of Tasks can increase as the problem size does
- ✓ Number of Tasks as small as possible but at least as large as the number of available processors
- ✓ Trade-off between agglomeration and cost of modifications to sequential codes is reasonable

Step 4: Mapping
Assigning Tasks to Processors



- ✓ **Mapping based on one task per processor and multiple tasks per processor have been considered**
- ✓ **Both static and dynamic allocation of tasks to processors have been evaluated**
- **(NA) If a dynamic allocation of tasks to processors is chosen, the Task allocator is not a bottleneck**
- ✓ **If Static allocation of tasks to processors is chosen, the ratio of tasks to processors is at least 10 to 1**

What's Missing?

- **The Performance Testing!**
  - In 1990 this code ran on a 120 Mflop Cray XMP with a grid of 20x50
  - Today even 100 time steps with a 100x100 grid runs in ~27 seconds on 1 GHz Pentium III
- **Is this a problem worth parallelizing?**
  - Maybe….
  - If the Grid were much bigger.
  - If the amount of computation per task increased significantly to model something more sophisticated.
- **The code certainly has good flexibility with regards to scaling.**

Heat2D Parallel Implementation
Parallel C Solution 1 of 7

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#define EPSILON 0.00001
#define N 100
#define time_steps 100

int main (int argc, char *argv[])
 {
  int i,j;
  int myid, numprocs;
  int from,to;
  int step;
  double time;
  double eps, enew, global_eps;
  double time_max = 3.0;
  double alpha = 0.06;
  double dx = 1.0/N;
  double dy = 1.0/time_steps;
  double dt = time_max/time_steps;
  double dxinv = 1.0/dx;
  double dyinv = 1.0/dy;
  double dtinv = 1.0/dt;
  double divinv = 1.0/(dtinv + 2 * alpha * (dxinv * dxinv + dyinv * dyinv));
```

```
double **t,*tS;
double **told,*toldS;
double **tstep,*tstepS;
double *sendR1,*sendR2;
double *recvR1,*recvR2;
double minval, maxval;
long clock(),cputime;
char fname[40];
FILE *out;
MPI_Status status;
```

Heat2D Parallel Implementation
Parallel C Solution 2 of 7

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

if (((N % numprocs) == 0) && (numprocs > 1))
 {
  if (myid == 0)
   {
    clock();
    tstepS = (double *)  malloc(N * N * sizeof(double));
    tstep =  (double **) malloc(N * sizeof(double *));

    for (i=0; i<N; i++)
     tstep[i] = &tstepS[i*N];
   }
  //allocate enough memory for each fraction of the mesh
  tS =    (double *)  malloc(((N/numprocs)+2) * N * sizeof(double));
  toldS = (double *)  malloc(((N/numprocs)+2) * N * sizeof(double));
  t =     (double **) malloc(((N/numprocs)+2) * sizeof(double *));
  told =  (double **) malloc(((N/numprocs)+2) * sizeof(double *));

  printf("rows = %d\n", (N/numprocs)+2);
  for (i=0; i<(N/numprocs)+2; i++)
   {
    t[i] = &tS[i*N];
    told[i] = &toldS[i*N];
   }
```

Heat2D Parallel Implementation
Parallel C Solution 3 of 7

```
 // set initial boundary conditions
  for (i=0; i<(N/numprocs)+2; i++)
   for (j=0; j<N; j++)
    told[i][j] = 0.0;

// for all time steps
  for (step = 1; step <= time_steps; step++)
   {
    time = step * (time_max/time_steps);
    // reset top boundary condition each timestep
    if (myid == 0)
     for (j=0; j<N; j++)
       told[0][j] = 2.0 * sin(time);
    do
     {
      // exchange the rows you need to share with other processes
      if (myid == 0)
       {
        //allocate memory for message arrays
        sendR1 = (double *) malloc(N * sizeof(double));
        recvR1 = (double *) malloc(N * sizeof(double));
        //send to my last computed row to myid+1
        for(j=0; j<N; j++) sendR1[j]=told[(N/numprocs)][j];
        MPI_Send(sendR1,N,MPI_DOUBLE,myid+1,0,MPI_COMM_WORLD);
        //receive my last row from myid+1
        MPI_Recv(recvR1,N,MPI_DOUBLE,myid+1,0,MPI_COMM_WORLD,&status);
        for(j=0; j<N; j++) told[(N/numprocs)+1][j]=recvR1[j];
        free(sendR1);
        free(recvR1);
       }
```

Heat2D Parallel Implementation
Parallel C Solution 4 of 7

```
else if ((myid > 0) && (myid < (numprocs-1)))
 {
  //allocate memory for message arrays
  sendR1 = (double *) malloc(N * sizeof(double));
  recvR1 = (double *) malloc(N * sizeof(double));
  sendR2 = (double *) malloc(N * sizeof(double));
  recvR2 = (double *) malloc(N * sizeof(double));
  //send my first computed row to myid-1
  for(j=0; j<N; j++) sendR1[j]=told[1][j];
  MPI_Send(sendR1,N,MPI_DOUBLE,myid-1,0,MPI_COMM_WORLD);
  //send my last computed row to myid+1
  for(j=0; j<N; j++) sendR2[j]=told[(N/numprocs)][j];
  MPI_Send(sendR2,N,MPI_DOUBLE,myid+1,0,MPI_COMM_WORLD);
  //receive my first row from myid-1
  MPI_Recv(recvR1,N,MPI_DOUBLE,myid-1,0,MPI_COMM_WORLD,&status);
  for(j=0; j<N; j++) told[0][j]=recvR1[j];
  //receive my last row from myid+1
  MPI_Recv(recvR2,N,MPI_DOUBLE,myid+1,0,MPI_COMM_WORLD,&status);
  for(j=0; j<N; j++) told[(N/numprocs)+1][j]=recvR2[j];
  free(sendR1);
  free(sendR2);
  free(recvR1);
  free(recvR2);
 }
```

Heat2D Parallel Implementation
Parallel C Solution 5 of 7

```
   if (myid == (numprocs-1))
     {
      //allocate memory for message arrays
      sendR1 = (double *) malloc(N * sizeof(double));
      recvR1 = (double *) malloc(N * sizeof(double));
      //send my first computed row myid-1
      for(j=0; j<N; j++) sendR1[j]=told[1][j];
      MPI_Send(sendR1,N,MPI_DOUBLE,myid-1,0,MPI_COMM_WORLD);
      //receive my first row myid-1
      MPI_Recv(recvR1,N,MPI_DOUBLE,myid-1,0,MPI_COMM_WORLD,&status);
      for(j=0; j<N; j++) told[0][j]=recvR1[j];
      free(sendR1);
      free(recvR1);
     }
    eps = 0.0;
    for (i=1; i<=(N/numprocs); i++)
      for (j=1; j<(N-1); j++)
      t[i][j]=((told[i][j+1]+told[i][j-1])*alpha*dyinv*dyinv+(told[i+1][j]+told[i-1][j])*
alpha*dxinv*dxinv+(told[i][j]*dtinv))*divinv;
    for (i=1; i<=(N/numprocs); i++)
     {
      for (j=1; j<(N-1); j++)
       {
        enew = fabs(t[i][j] - told[i][j]);
        if (enew > eps) { eps = enew; }
       }
     }
    for (i=0; i<((N/numprocs)+2); i++)
     for (j=0; j<N; j++)
      told[i][j] = t[i][j];
```

Heat2D Parallel Implementation
Parallel C Solution 6 of 7

```
 MPI_Allreduce(&eps,&global_eps,1,MPI_DOUBLE,MPI_MAX,MPI_COMM_WORLD);
 }
while(global_eps > EPSILON);

// gather all the subsets of the grid
MPI_Gather(told[1],N*(N/numprocs),MPI_DOUBLE,tstepS,N*(N/numprocs),MPI_DOUBLE,0,MPI_COMM_WORLD);
// Produce a raster image for this time step
if (myid == 0)
 {
  minval = 0.0;
  maxval = 0.0;
  for (i=0; i<N; i++)
   {
    for (j=0; j<N; j++)
     {
      if (tstep[i][j] < minval) { minval = tstep[i][j]; }
      if (tstep[i][j] > maxval) { maxval = tstep[i][j]; }
     }
   }
  sprintf(fname,"Output\\heat%03d.raw",step);
  out = fopen(fname,"w+b");
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      fprintf(out,"%c",(int)(((tstep[i][j]-minval)*255.0)/(maxval - minval)));
  fclose(out);
  printf("Time step: %d\r",step);
 }
 MPI_Barrier(MPI_COMM_WORLD);
} // for all time steps
```

Heat2D Parallel Implementation
Parallel C Solution 7 of 7

```
    free(t);
    free(tS);
    free(told);
    free(toldS);
    if (myid == 0)
     {
      free(tstep);
      free(tstepS);
      cputime = clock();
      printf("%d time steps in %.2f seconds\n",step-1,cputime/1.0e+3);
     }
    } // if ((N % numprocs) == 0))
   else
    if (myid == 0)
     {
      if (numprocs < 2)
       printf("ERROR: Must have at least 2 processes\n");
      if ((N % numprocs) != 0)
       printf("ERROR: Grid must be evenly divisible by the number of
   processes\n");
     }
   MPI_Finalize();
}
```

Using Simple Visualization for Debugging