Lecture 2

MPI, OpenMP, MATLAB*P

A parallel language must provide mechanisms for implementing parallel algorithms, i.e., to specify various levels of parallelism and define parallel data structures for distributing and sharing information among processors.

Most current parallel languages add parallel constructs for standard sequential languages. Different parallel languages provide different basic constructs. The choice largely depends on the parallel computing model the language means to support.

There are at least three basic parallel computation models other than vector and pipeline model: *data parallel*, *message passing*, and *shared memory task parallel*.

2.1 Programming style

Data parallel vs. message passing. Explicit communication can be somewhat hidden if one wants to program in a data parallel style; it's something like SIMD: you specify a single action to execute on all processors. Example: if `A`, `B` and `C` are matrices, you can write `C=A+B` and let the compiler do the hard work of accessing remote memory locations, partition the matrix among the processors, etc.

By contrast, explicit message passing gives the programmer careful control over the communication, but programming requires a lot of knowledge and is much more difficult (as you probably understood from the previous section).

There are uncountable other alternatives, still academic at this point (in the sense that no commercial machine comes with them bundled). A great deal of research has been conducted on multithreading; the idea is that the programmer expresses the computation graph in some appropriate language and the machine executes the graph, and potentially independent nodes of the graph can be executed in parallel. Example in Cilk (multithreaded C, developed at MIT by Charles Leiserson and his students):

```
thread int fib(int n)
{
    if (n<2)
        return n;
    else {
        cont int x, y;
        x = spawn fib (n-2);
        y = spawn fib (n-1);
```

```
        sync;
        return x+y;
    }
}
```

Actually Cilk is a little bit different right now, but this is the way the program will look like when you read these notes. The whole point is that the two computations of `fib(n-1)` and `fib(n-2)` can be executed in parallel. As you might have expected, there are dozens of multithreaded languages (functional, imperative, declarative) and implementation techniques; in some implementations the thread size is a single-instruction long, and special processors execute this kind of programs. Ask Arvind at LCS for details.

Writing a good HPF compiler is difficult and not every manufacturer provides one; actually for some time TMC machines were the only machines available with it. The first HPF compiler for the Intel Paragon dates December 1994.

Why SIMD is not necessarily the right model for data parallel programming. Consider the following Fortran fragment, where `x` and `y` are vectors:

```
where (x > 0)
     y = x + 2
elsewhere
     y = -x + 5
endwhere
```

A SIMD machine might execute both cases, and discard one of the results; it does twice the needed work (see why? there is a single flow of instructions). This is how the CM-2 operated.

On the other hand, an HPF compiler for a SIMD machine can take advantage of the fact that there will be many more elements of `x` than processors. It can execute the `where` branch until there are no positive elements of `x` that haven't been seen, then it can execute the `elsewhere` branch until all other elements of `x` are covered. It can do this provided the machine has the ability to generate independent memory addresses on every processor.

Moral: even if the programming model is that there is one processor per data element, the programmer (and the compiler writer) must be aware that it's not true.

2.2 Message Passing

In the SISD, SIMD, and MIMD computer taxonomy, SISD machines are conventional uniprocessors, SIMD are single instruction stream machines that operate in parallel on ensembles of data, like arrays or vectors, and MIMD machines have multiple active threads of control (processes) that can operate on multiple data in parallel. How do these threads share data and how do they synchronize? For example, suppose two processes have a producer/consumer relationship. The producer generates a sequence of data items that are passed to the consumer, which processes them further. How does the producer deliver the data to the consumer?

If they share a common memory, then they agree on a location to be used for the transfer. In addition, they have a mechanism that allows the consumer to know when that location is full, *i.e.* it has a valid datum placed there for it by the producer, and a mechanism to read that location and change its state to empty. A full/empty bit is often associated with the location for this purpose. The hardware feature that is often used to do this is a "test-and-set" instruction that tests a bit in memory and simultaneously sets it to one. The producer has the obvious dual mechanisms.

Many highly parallel machines have been, and still are, just collections of independent computers on some sort of a network. Such machines can be made to have just about any data sharing and synchronization mechanism; it just depends on what software is provided by the operating system, the compilers, and the runtime libraries. One possibility, the one used by the first of these machines (The Caltech Cosmic Cube, from around 1984) is message passing. (So it's misleading to call these "message passing machines"; they are really multicomputers with message passing library software.)

From the point of view of the application, these computers can send a message to another computer and can receive such messages off the network. Thus, a process cannot touch any data other than what is in its own, private memory. The way it communicates is to send messages to and receive messages from other processes. Synchronization happens as part of the process, by virtue of the fact that both the sending and receiving process have to make a call to the system in order to move the data: the sender won't call send until its data is already in the send buffer, and the receiver calls receive when its receive buffer is empty and it needs more data to proceed.

Message passing systems have been around since the Cosmic Cube, about ten years. In that time, there has been a lot of evolution, improved efficiency, better software engineering, improved functionality. Many variants were developed by users, computer vendors, and independent software companies. Finally, in 1993, a standardization effort was attempted, and the result is the Message Passing Interface (MPI) standard. MPI is flexible and general, has good implementations on all the machines one is likely to use, and is almost certain to be around for quite some time. We'll use MPI in the course. On one hand, MPI is complicated considering that there are more than 150 functions and the number is still growing. But on the other hand, MPI is simple because there are only six basic functions: `MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Send` and `MPI_Recv`.

In print, the best MPI reference is the handbook *Using MPI*, by William Gropp, Ewing Lusk, and Anthony Skjellum, published by MIT Press ISBN 0-262-57104-8.

The standard is on the World WIde Web. The URL is
http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html. An updated version is at
ftp://ftp.mcs.anl.gov/pub/mpi/mpi-1.jun95/mpi-report.ps

2.2.1 Who am I?

On the SP-2 and other multicomputers, one usually writes one program which runs on all the processors. In order to differentiate its behavior, (like producer and consumer) a process usually first finds out at runtime its rank within its process group, then branches accordingly. The calls

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

sets `size` to the number of processes in the group specified by `comm` and the call

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

sets `rank` to the rank of the calling process within the group (from 0 up to $n-1$ where n is the size of the group). Usually, the first thing a program does is to call these using `MPI_COMM_WORLD` as the communicator, in order to find out the answer to the *big* questions, "Who am I?" and "How many other 'I's are there?".

Okay, I lied. That's the second thing a program does. Before it can do anything else, it has to make the call

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

where argc and argv should be pointers to the arguments provided by UNIX to main(). While we're at it, let's not forget that one's code needs to start with

```
#include "mpi.h"
```

The *last* thing the MPI code does should be

```
MPI_Finalize()
```

No arguments.

Here's an MPI multi-process "Hello World":

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

main(int argc, char** argv) {
  int i, myrank, nprocs;
  double a = 0, b = 1.1, c = 0.90;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  printf("Hello world!  This is process %d out of %d\n",myrank, nprocs);
  if (myrank == 0) printf("Some processes are more equal than others.");
  MPI_Finalize();
}  /*  main  */
```

which is compiled and executed on the SP-2 at Ames by

```
babbage1% mpicc -O3 example.c
babbage1% a.out -procs 2
```

and produces (on the standard output)

```
0:Hello world!  This is process 0 out of 2
1:Hello world!  This is process 1 out of 2
0:Some processes are more equal than others.
```

Another important thing to know about is the MPI wall clock timer:

```
double MPI_Wtime()
```

which returns the time in seconds from some unspecified point in the past.

2.2.2 Sending and receiving

In order to get started, let's look at the two most important MPI functions, MPI_Send and MPI_Recv.

The call

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

sends `count` items of data of type `datatype` starting at the location `buf`. In all message passing systems, the processes have identifiers of some kind. In MPI, the process is identified by its `rank`, an integer. The data is sent to the processes whose rank is `dest`. Possible values for `datatype` are `MPI_INT, MPI_DOUBLE, MPI_CHAR` *etc.* `tag` is an integer used by the programmer to allow the receiver to select from among several arriving messages in the `MPI_Recv`. Finally, `comm` is something called a communicator, which is essentially a subset of the processes. Ordinarily, message passing occurs within a single subset. The subset `MPI_COMM_WORLD` consists of all the processes in a single parallel job, and is predefined.

A receive call matching the send above is

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

`buf` is where the data is placed once it arrives. `count`, an input argument, is the size of the buffer; the message is truncated if it is longer than the buffer. Of course, the receive has to be executed by the correct destination process, as specified in the `dest` part of the send, for it to match. `source` must be the rank of the sending process. The communicator and the tag must match. So must the datatype.

The purpose of the datatype field is to allow MPI to be used with heterogeneous hardware. A process running on a little-endian machine may communicate integers to another process on a big-endian machine; MPI converts them automatically. The same holds for different floating point formats. Type conversion, however, is not supported: an integer must be sent to an integer, a double to a double, *etc.*

Suppose the producer and consumer transact business in two word integer packets. The producer is process 0 and the consumer is process 1. Then the send would look like this:

```
int outgoing[2];
MPI_Send(outgoing, 2, MPI_INT, 1 100, MPI_COMM_WORLD)
```

and the receive like this:

```
MPI_Status stat;
int incoming[2];
MPI_Recv(incoming, 2, MPI_INT, 0 100, MPI_COMM_WORLD, &stat)
```

What if one wants a process to which several other processes can send messages, with service provided on a first-arrived, first-served basis? For this purpose, we don't want to specify the source in our receive, and we use the value `MPI_ANY_SOURCE` instead of an explicit source. The same is true if we want to ignore the tag: use `MPI_ANY_TAG`. The basic purpose of the status argument, which is an output argument, is to find out what the tag and source of such a received message are. `status.MPI_TAG` and `status.MPI_SOURCE` are components of the struct `status` of type int that contain this information after the `MPI_Recv` function returns.

This form of send and receive are "blocking", which is a technical term that has the following meaning. for the send, it means that `buf` has been read by the system and the data has been moved out as soon as the send returns. The sending process can write into it without corrupting the message that was sent. For the receive, it means that `buf` has been filled with data on return. (A call to `MPI_Recv` with no corresponding call to `MPI_Send` occurring elsewhere is a very good and often used method for hanging a message passing application.)

MPI implementations may use buffering to accomplish this. When send is called, the data are copied into a system buffer and control returns to the caller. A separate system process (perhaps using communication hardware) completes the job of sending the data to the receiver. Another implementation is to wait until a corresponding receive is posted by the destination process, then transfer the data to the receive buffer, and finally return control to the caller. MPI provides two variants of send, `MPI_Bsend` and `MPI_Ssend` that force the buffered or the rendezvous implementation. Lastly, there is a version of send that works in “ready” mode. For such a send, the corresponding receive must have been executed previously, otherwise an error occurs. On some systems, this may be faster than the blocking versions of send. All four versions of send have the same calling sequence.

NOTE: MPI allows a process to send itself data. Don’t try it. On the SP-2, if the message is big enough, it doesn’t work. Here’s why. Consider this code:

```
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Send(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat);
```

The programmer is attempting to send data from process zero to all processes, including process zero; $4 \cdot \text{count}$ bytes of it. If the system has enough buffer space for the outgoing messages, this succeeds, but if it doesn’t, then the send blocks until the receive is executed. But since control does not return from the blocking send to process zero, the receive never does execute. If the programmer uses buffered send, then this deadlock cannot occur. An error will occur if the system runs out of buffer space, however:

```
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Bsend(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat);
```

2.2.3 Tags and communicators

Tags are used to keep messages straight. An example will illustrate how they are used. Suppose each process in a group has one integer and one real value, and we wish to find, on process zero, the sum of the integers and the sum of the reals. Lets write this:

```
itag = 100;
MPI_Send(&intvar, 1, MPI_INT, 0, itag, MPI_COMM_WORLD);
ftag = 101;
MPI_Send(&floatvar, 1, MPI_FLOAT, 0, ftag, MPI_COMM_WORLD);
/****  Sends are done.   Receive on process zero   ****/
if (myrank == 0) {
   intsum = 0;
   for (kount = 0; kount < nprocs; kount++) {
      MPI_Recv(&intrecv, 1, MPI_INT, MPI_ANY_SOURCE, itag, MPI_COMM_WORLD, &stat);
      intsum += intrecv;
   }
   fltsum = 0;
   for (kount = 0; kount < nprocs; kount++) {
      MPI_Recv(&fltrecv, 1, MPI_FLOAT, MPI_ANY_SOURCE, ftag, MPI_COMM_WORLD, &stat);
      fltsum += fltrecv;
   }
```

```
}
```

It looks simple, but there are a lot of subtleties here! First, note the use of MPI_ANY_SOURCE in the receives. We're happy to receive the data in the order it arrives. Second, note that we use two different tag values to distinguish between the int and the float data. Why isn't the MPI_TYPE filed enough? Because MPI does not include the type as part of the message "envelope". The envelope consists of the source, destination, tag, and communicator, and these must match in a send-receive pair. Now the two messages sent to process zero from some other process are guaranteed to arrive in the order they were sent, namely the integer message first. But that does not mean that all of the integer message precede all of the float messages! So the tag is needed to distinguish them.

This solution creates a problem. Our code, as it is now written, sends off a lot of messages with tags 100 and 101, then does the receives (at process zero). Suppose we called a library routine written by another user before we did the receives. What if that library code uses the same message tags? Chaos results. We've "polluted" the tag space. Note, by the way, that synchronizing the processes before calling the library code does not solve this problem.

MPI provides communicators as a way to prevent this problem. The communicator is a part of the message envelope. So we need to change communicators while in the library routine. To do this, we use `MPI_Comm_dup`, which makes a new communicator with the same processes in the same order as an existing communicator. For example

```
void safe_library_routine(MPI_Comm oldcomm)
{
   MPI_Comm mycomm;
   MPI_Comm_dup(oldcomm, &mycomm);
   <library code using mycomm for communication>
   MPI_Comm_free(&mycomm);
}
```

The messages sent and received inside the library code cannot interfere with those send outside.

2.2.4 Performance, and tolerance

Try this exercise. See how long a message of length n bytes takes between the call time the send calls send and the time the receiver returns from receive. Do an experiment and vary n. Also vary the rank of the receiver for a fixed sender. Does the model

$$\mathrm{Elapsed\_Time}(n, r) = \alpha + \beta n$$

work? (r is the receiver, and according to this model, the cost is receiver independent.)

In such a model, the latency for a message is α seconds, and the bandwidth is $1/\beta$ bytes/second. Other models try to split α into two components. The first is the time actually spent by the sending processor and the receiving processor on behalf of a message. (Some of the per-byte cost is also attributed to the processors.) This is called the overhead. The remaining component of latency is the delay as the message actually travels through the machines interconnect network. It is ordinarily much smaller than the overhead on modern multicomputers (ones, rather than tens of microseconds).

A lot has been made about the possibility of improving performance by "tolerating" communication latency. To do so, one finds other work for the processor to do while it waits for a message to arrive. The simplest thing is for the programmer to do this explicitly. For this purpose, there are "nonblocking" versions of send and receive in MPI and other dialects.

Nonblocking send and receive work this way. A nonblock **send start** call initiates a send but returns before the data are out of the send buffer. A separate call to **send complete** then blocks the sending process, returning only when the data are out of the buffer. The same two-phase protocol is used for nonblocking receive. The **receive start** call returns right away, and the **receive complete** call returns only when the data are in the buffer.

The simplest mechanism is to match a nonblocking receive with a blocking send. To illustrate, we perform the communication operation of the previous section using nonblocking receive.

```
MPI_Request request;
MPI_IRecv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &request);
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Send(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Wait(&request, &stat);
```

`MPI_Wait` blocks until the nonblocking operation identified by the handle `request` completes. This code is correct regardless of the availability of buffers. The sends will either buffer or block until the corresponding **receive start** is executed, and all of these will be.

Before embarking on an effort to improve performance this way, one should first consider what the payoff will be. In general, the best that can be achieved is a two-fold improvement. Often, for large problems, it's the bandwidth (the βn term) that dominates, and latency tolerance doesn't help with this. Rearrangement of the data and computation to avoid some of the communication altogether is required to reduce the bandwidth component of communication time.

2.2.5 Who's got the floor?

We usually think of send and receive as the basic message passing mechanism. But they're not the whole story by a long shot. If we wrote codes that had genuinely different, independent, asynchronous processes that interacted in response to random events, then send and receive would be used to do all the work. Now consider computing a dot product of two identically distributed vectors. Each processor does a local dot product of its pieces, producing one scalar value per processor. Then we need to add them together and, probably, broadcast the result. Can we do this with send and receive? Sure. Do we want to? No. No because it would be a pain in the neck to write and because the MPI system implementor may be able to provide the two necessary, and generally useful collective operations (sum, and broadcast) for us in a more efficient implementation.

MPI has lots of these "collective communication" functions. (And like the sum operation, they often do computation as part of the communication.)

Here's a sum operation, on doubles. The variable `sum` on process `root` gets the sum of the variables `x` on all the processes.

```
double x, sum;
int root, count = 1;
MPI_Reduce(&x, &sum, count, MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD);
```

The fifth argument specifies the operation; other possibilities are `MPI_MAX, MPI_LAND, MPI_BOR,` ... which specify maximum, logical AND, and bitwise OR, for example.

The semantics of the collective communication calls are subtle to this extent: nothing happens except that a process stops when it reaches such a call, until *all* processes in the specified group reach it. Then the reduction operation occurs and the result is placed in the `sum` variable on the *root* processor. Thus, reductions provide what is called a *barrier* synchronization.

There are quite a few collective communication operations provided by MPI, all of them useful and important. We will use several in the assignment. To mention a few, MPI_Bcast broadcasts a vector from one process to the rest of its process group; MPI_Scatter sends different data from one process to each process in its a group; MPI_Gather is the inverse of a scatter: one process receives and concatenates data from all processes in its group; MPI_Allgather is like a gather followed by a broadcast: all processes receive the concatenation of data that are initially distributed among them; MPI_Reduce_scatter is like reduce followed by scatter: the result of the reduction ends up distributed among the process group. Finally, `MPI_Alltoall` implements a very general communication in which each process has a separate message to send to each member of the process group.

Often the process group in a collective communication is some subset of all the processors. In a typical situation, we may view the processes as forming a grid, let's say a 2d grid, for example. We may want to do a reduction operation within rows of the process grid. For this purpose, we can use MPI_Reduce, with a separate communicator for each row.

To make this work, each process first computes its coordinates in the process grid. MPI makes this easy, with

```
int nprocs, myproc, procdims[2], myproc_row, myproc_col;
MPI_Dims_create(nprocs, 2, procdims);
myproc_row = myrank / procdims[1];
myproc_col = myrank % procdims[1];
```

Next. one creates new communicators, one for each process row and one for each process column. The calls

```
MPI_Comm my_prow, my_pcol;
MPI_Comm_split(MPI_COMM_WORLD, myproc_row, 0, &my_prow);
MPI_Comm_split(MPI_COMM_WORLD, myproc_row, 0, &my_prow);
```

create them and

```
MPI_Comm_free(&my_prow);
MPI_Comm_free(&my_pcol);
```

free them. The reduce-in-rows call is then

```
MPI_Reduce(&x, &sum, count, MPI_DOUBLE, MPI_SUM, 0, my_prow);
```

which leaves the sum of the vectors `x` in the vector `sum` in the process whose rank in the group is zero: this will be the first process in the row. The general form is

```
MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm newcomm)
```

As in the example above, the group associated with `comm` is split into disjoint subgroups, one for every different value of `color`; the communicator for the subgroup that this process belongs to is returned in `newcomm`. The argument `key` determines the rank of this process within `newcomm`; the members are ranked according to their value of `key`, with ties broken using the rank in `comm`.

2.3 More on Message Passing

2.3.1 Nomenclature

- Performance:

  Latency: the time it takes to send a zero length message (overhead)
  Bandwidth: the rate at which data can be sent

- Synchrony:

  Synchronous: sending function returns when a matching receiving operation has been initiated at the destination.
  Blocking: sending function returns when the message has been safely copied.
  Non-blocking (asynchronous): sending function returns immediately. Message buffer must not be changed until it is safe to do so

- Miscellaneous:

  Interrupts: if enabled, the arrival of a message interrupts the receiving processor
  Polling: the act of checking the occurrence of an event
  Handlers: code written to execute when an interrupt occurs
  Critical sections: sections of the code which cannot be interrupted safely
  Scheduling: the process of determining which code to run at a given time
  Priority: a relative statement about the ranking of an object according to some metric

2.3.2 The Development of Message Passing

- Early Multicomputers:
  UNIVAC
  Goodyear MPP (SIMD.)
  Denelcor HEP (Shared memory)

- The Internet and Berkeley Unix (High latency, low bandwidth)

  Support for communications between computers is part of the operating system (sockets, ports,remote devices)
  Client/server applications appear

- Parallel Machines (lowest latency,high bandwidth - dedicated networks)

  Ncube
  Intel (Hypercubes, Paragon)
  Meiko (CS1,CS2)
  TMC (CM-5)

- Workstation clusters (lower latency, high bandwidth, popular networks)

  IBM (SP1,2) - optional proprietary network

- Software (libraries):

  CrOS, CUBIX, NX, Express
  Vertex
  EUI/MPL (adjusts to the machine operating system)
  CMMD

PVM (supports heterogeneous machines; dynamic machine configuration)
PARMACS, P4, Chamaleon
MPI (analgum of everything above and adjusts to the operating system)

- Systems:

  Mach
  Linda (object-based system)

2.3.3 Machine Characteristics

- Nodes:

  CPU, local memory,perhaps local I/O

- Networks:

  Topology: Hypercube,Mesh,Fat-Tree, other
  Routing: circuit, packet, wormhole, virtual channel random
  Bisection bandwidth (how much data can be sent along the net)
  Reliable delivery
  Flow Control
  "State" : Space sharing, timesharing, stateless

- Network interfaces: Dumb: Fifo, control registers
  Smart: DMA (Direct Memory Access) controllers
  Very Smart: handle protocol managememt

2.3.4 Active Messages

[Not yet written]

2.4 OpenMP for Shared Memory Parallel Programming

OpenMP is the current industry standard for shared-memory parallel programming directives. Jointly defined by a group of major computer hardware and software vendors, OpenMP Application Program Interface (API) supports shared-memory parallel programming in C/C++ and Fortran on Unix and Windows NT platforms. The OpenMP specifications are owned and managed by the OpenMP Architecture Review Board (ARB). For detailed information about OpenMP, we refer readers to its website at

http://www.openmp.org/.

OpenMP stands for Open specifications for Multi Processing. It specify a set of compiler directives, library routines and environment variables as an API for writing multi-thread applications in C/C++ and Fortran. With these directives or pragmas, multi-thread codes are generated by compilers. OpenMP is a shared momoery model. Threads communicate by sharing variables. The cost of communication comes from the synchronization of data sharing in order to protect data conficts.

Compiler pragmas in OpenMP take the following forms:

- for C and C++,

`#pragma omp` *construct [clause [clause]...]*,

- for Fortran,

  `C$OMP` *construct [clause [clause]...]*
  `!$OMP` *construct [clause [clause]...]*
  `*$OMP` *construct [clause [clause]...]*.

For example, OpenMP is usually used to parallelize loops, a simple parallel C program with its loops split up looks like

```
void main()
{
  double Data[10000];
#pragma omp parallel for
  for (int i=0; i<10000; i++) {
    task(Data[i]);
  }
}
```

If all the OpenMP constructs in a program are compiler pragmas, then this program can be compiled by compilers that do not support OpenMP.

OpenMP's constructs fall into 5 categories, which are briefly introduced as the following:

1. **Parallel Regions**
   Threads are created with the "omp parallel" pragma. In the following example, a 8-thread *parallel region* is created:

   ```
   double Data[10000];
   omp_set_num_threads(8);
   #pragma omp parallel
   {
     int ID = omp_get_thread_num();
     task(ID,Data);
   }
   ```

   Each thread calls `task(ID,Data)` for `ID` = 0 to 7.

2. **Work Sharing**
   The "for" work-sharing construct splits up loop iterations among the threads a parallel region. The pragma used in our first example is the short hand form of "omp parallel" and "omp for" pragmas:

   ```
   void main()
   {
     double Data[10000];
   #pragma omp parallel
   #pragma omp for
     for (int i=0; i<10000; i++) {
       task(Data[i]);
     }
   }
   ```

"Sections" is another work-sharing construct which assigns different jobs (different pieces of code) to each thread in a parallel region. For example,

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      job1();
    }
    #pragma omp section
    {
      job2();
    }
  }
}
```

Similarly, there is also a "parallel sections" construct.

3. **Data Environment**
   In the shared-memory programming model, global variables are shared among threads, which are file scope and static variables for C, and common blocks, save and module variables for Fortran. Automatic variables within a statement block are private, also stack variables in sub-programs called from parallel regions. Constructs and clauses are available to selectively change storage attributes.

   - The "shared" clause uses a shared memory model for the variable, that is, all threads share the same variable.
   - The "private" clause gives a local copy of the variable in each thread.
   - "firstprivate" is like "private", but the variable is initialized with its value before the thread creation.
   - "lastprivate" is like "private", but the value is passed to a global variable after the thread execution.

4. **Synchronization**
   The following constructs are used to support synchronization:

   - "critical" and "end critical" constructs define a *critical region*, where only one thread can enter at a time.
   - "atomic" construct defines a critical region that only contains one simple statement.
   - "barrier" construct is usually implicit, for example at the end of a parallel region or at the end of a "for" work-sharing construct. The barrier construct makes threads wait until all of them arrive.
   - "ordered" construct enforces the sequential order for a block of code.
   - "master" construct marks a block of code to be only executed by the master thread. The other threads just skip it.

- "single" construct marks a block of code to be executed only by one thread.
- "flush" construct denotes a sequence point where a thread tries to create a consistent view of memory.

5. **Runtime functions and environment variables**
   Besides constructs, there are clauses, library routines and environment variables in OpenMP. We list a few of the most important below, please refer to OpenMP's web site for more details.

   - The number of threads can be controlled using the runtime environment routines "omp_set_num_threads()", "omp_get_num_threads()", "omp_get_thread_num()", "omp_get_max_threads()".
   - The number of processors in the system is given by "omp_num_procs()".

2.5 STARP

Star-P is a set of extensions to Matlab aimed at making it simple to parallelize many common computations. A running instance of Star-P consists of a control Matlab process (connected to the Matlab notebook/desktop the user sees) and a slaved Matlab processes for each processor available to the system. It permits a matlab user to declare distributed matrix objects, whose contents are distributed in various ways across the processors available to Star-P, and to direct each processor to apply a matlab function to the portion of a distributed matrix it stores. It also supplies parallel versions of many of the dense (and some of the sparse) matrix operations which occur in scientific computing, such as eigenvector/eigenvalue computations.

This makes Star-P especially convenient for embarassingly parallel problems, as all one needs to do is:

1. Write a matlab function which carries out a part of the embarassingly parallel problem, parametrized such that it can take the inputs it needs as the rows or columns of a matrix.
2. Construct a distributed matrix containing the parameters needed for each slave process.
3. Tell each slave to apply the function to the portion of the matrix it has.
4. Combine the results from each process.

Now, some details. Star-P defines the variable `np` to store the number of slave processes it controls. On a machine with two apparent processors, for example, `np` would equal 2. Distributed matrices are declared in Star-P by appending `*p` to the dimension along which you want your matrix to be distributed. For example, the code below declares `A` to be a row-distributed matrix; each processor theoretically gets an equal chunk of rows from the matrix.

```
A = ones(100*p, 100);
```

To declare a column-distributed matrix, one simply does:

```
A = ones(100, 100*p);
```

Beware: if the number of processors you're working with does not evenly divide the size of the dimension you are distributing along, you may encounter unexpected results. Star-P also supports matrices which are distributed into blocks, by appending `*p` to both dimensions; we will not go into the details of this here.

After declaring `A` as a distributed matrix, simply evaluating it will yield something like:

```
A = ddense object: 100-by-100
```

This is because the elements of `A` are stored in the slave processors. To bring the contents to the control process, simply index A. For example, if you wanted to view the entire contents, you'd do `A(:,:)`.

To apply a Matlab function to the chunks of a distributed matrix, use the `mm` command. It takes a string containing a procedure name and a list of arguments, each of which may or may not be distributed. It orders each slave to apply the function to the chunk of each distributed argument (echoing non-distributed arguments) and returns a matrix containing the results of each appended together. For example, `mm('fft', A)` (with `A` defined as a 100-by-100 column distributed matrix) would apply the `fft` function to each of the 25-column chunks. Beware: the chunks must each be distributed in the same way, and the function must return chunks of the same size. Also beware: `mm` is meant to apply a function to chunks. If you want to compute the two-dimensional `fft` of `A` in parallel, do not use `mm('fft2', A)`; that will compute (in serial) the `fft2`s of each chunk of A and return them in a matrix. `eig(A)`, on the other hand, will apply the parallel algorithm for eigenstuff to `A`.

Communication between processors must be mediated by the control process. This incurs substantial communications overhead, as the information must first be moved to the control process, processed, then sent back. It also necessitates the use of some unusual programming idioms; one common pattern is to break up a parallel computation into steps, call each step using `mm`, then do matrix column or row swapping in the control process on the distributed matrix to move information between the processors. For example, given a matrix `B = randn(10, 2*p)` (on a Star-P process with two slaves), the command `B = B(:,[2,1])` will swap elements between the two processors.

Some other things to be aware of:

1. Star-P provides its functionality by overloading variables and functions from Matlab. This means that if you overwrite certain variable names (or define your own versions of certain functions), they will shadow the parallel versions. In particular, DO NOT declare a variable named `p`; if you do, instead of distributing matrices when `*p` is appended, you will multiply each element by your variable `p`.

2. `persistent` variables are often useful for keeping state across stepped computations. The first time the function is called, each persistent variable will be bound to the empty matrix `[]`; a simple `if` can test this and initialize it the first time around. Its value will then be stored across multiple invocations of the function. If you use this technique, make sure to `clear` those variables (or restart Star-P) to ensure that state isn't propagated across runs.

3. To execute a different computation depending on processor id, create a matrix `id = 1:np` and pass it as an argument to the function you call using `mm`. Each slave process will get a different value of `id` which it can use as a different unique identifier.

4. If `mm` doesn't appear to find your `m`-files, run the `mmpath` command (which takes one argument - the directory you want `mm` to search in).

Have fun!