A Tutorial Introduction to Charm

L.V. Kale
Department of Computer Science
University of Illinois at Urbana Champaign
1304 W. Springfield Ave., Urbana, IL-61801

1 Introduction

Charm is a machine independent parallel programming system. Programs written using this system will run unchanged on MIMD machines with or without a shared memory. Some systems provide machine independence by supporting a few low level mechanisms (e.g. universal message send/receive primitives). Charm can be thought as a layer on top of such machine independence mechanisms. It provides high-level mechanisms and strategies (such as dynamic load balancing strategies), to facilitate the task of developing even highly complex parallel applications.

The design of the system is based on the following tenets:

1. Locality of data is the cost measure that applies across all MIMD machines. The system design induces better data locality. This is how it can support machine independence without losing efficiency.

2. Latency of communication - the idea that remote data will take longer to access - is a significant issue common across most MIMD platforms. *Message-driven execution*, supported in Charm, is a very useful mechanism for tolerating or hiding this latency. In message driven execution (which is distinct from just message-passing), processor is allocated to a process only when a message for the process is received. This means when a process blocks, waiting for a message, another process may execute on the processor. It also means that a single process may block for any number of distinct messages, and will be awakened when any of these messages arrive. Thus, it forms an effective way of scheduling a processor in the presence of potentially large latencies.

3. Dynamic creation of work is necessary in many applications programs. The system supports this by providing dynamic (as well as static) load balancing strategies.

4. A major activity in a parallel computation is creation and sharing of information. Information is shared in many specific modes. The system provides six information sharing modes, each of which may be implemented differently and efficiently on different machines.

5. It should be possible to develop parallel software by reusing existing parallel software. The system supports this with a well-developed "module" construct, and associated mechanisms. Along with message-driven execution, these mechanisms allow for compositionality of modules without sacrificing the latency-tolerance.

The programs are written in C with a few syntactic extensions explained below. It is possible to interface to other languages such as FORTRAN using the foreign language interface that C provides.

In this tutorial, we will introduce the reader to programming with Charm via a series of examples, introducing various features as needed. The complete language description can be found in the Charm Language manual, and I suggest you have the manual handy for reference while reading the tutorial.

2 Stucture of the Program

A Charm program consists of one or more modules, with one module per file. Each module may include C type definitions, message declarations, definitions and declarations of specifically shared variables, C function definitions, *chare* definitions, and *branchOffice chare* definitions. It must also include the definition of a main chare.

A chare (French for "chore" - a word borrowed from Bob Keller and the REDIFLOW project) has similarities with a process, a concurrent object, an actor, an ADA task, etc. You may want to say "process" instead of chare until you get used to it. Note that there is a distinction between chare instances and a chare definition. A chare definition merely specifies the behavior of a class of possible chare instances. A chare instance is created by a system call.

A chare definition starts with the name of the chare, a declaration of local variables, and a collection of entry-point definitions. In addition, it may also include private functions, which are just like C functions, except they are allowed to access the local variables of the chare and they can only be called from within the chare. Each entry point definition includes a declaration of a message variable, and an arbitrary C-code-block. The C-code-block may contain Charm calls in addition to the usual C code. All Charm calls are non blocking. There are calls available for creating a new chare and for sending a message to a chare whose ID you know.

The Branch-office Chare (BOC) is a chare whose "branch" or representative exists on every processor. It has a similar syntax as chare, but in addition to the private functions, it may also include public functions, which can be called from outside the BOC - from another chare running on a particular processor, for example. All the branches are identified by a single instance ID, so when a chare makes a call to a public function of a BOC, it need not know which processor it is on. The call is automatically directed to the local branch of the BOC named in the call.

3 A Simple Program to Get Started

Program I shows a simple but complete parallel Charm program which finds all solutions to the 8 non-attacking queens problem. (The N non-attacking queens problem is to find a placement for N queens on a generalized NxN chess-board such that no two queens attack each other. A queen is a chess piece that is said to attack another queen if they are both in the same row, column, or diagonal as each other. I.e. if one queen is at position (x_1, y_1), and the other one is at (x_2, y_2), then they attack each other if either $x_1 = x_2$ or $y_1 = y_2$ or $\mid x_1 - x_2 \mid = \mid y_1 - y_2 \mid$.)

The first thing to notice about this program is its components are largely C code fragments. The program begins with constant and message-type definitions, followed by two chare definitions, and a few C functions definitions. Every Charm program must begin with type definitions, followed by the definition of the main chare, and then an arbitrary number of chare definitions.

The C function definitions must occur after type definitions, but may be interspersed between chare definitions.

The parallel algorithm is simple: start with an empty board, and place queens row by row. For any given partially filled board which has the first K rows filled, find positions in the next row that do not conflict with currently placed queens, and start a new chare for exploring each such position further. The algorithm requires a message type that describes a partially filled board, which is used to create a new chare. So, we declare a message type called `PartialBoard` with two fields. The main chare has an entry point called `CharmInit`, and this program's execution begins here. The code at that entry point allocates a message of type `PartialBoard` using the `CkAllocMsg` call, fills a field of this message (to indicate that no queens have been placed) and creates a chare of type `queens` with the initial message pointed to by `pMsg`. (`CkAllocMsg` is similar to malloc in usage, except it takes the type-name of the message instead of just its size. By the way, `malloc` should never be used in chare kernel programs. Instead, `CkAlloc` should be used to allocate memory, and `CkAllocMsg` to allocate messages.) The `queens` chare needs no local variable, since it is simply going to create children chares, if needed, and exit. `NewBoard` is the only entrypoint in this chare definition. When the execution begins at an instance of this chare at the `NewBoard` entrypoint, the variable `m` points to a message of this type. The code simply loops through all column positions in the `nextRow`, and if they are feasible positions, allocates a new partialBoard message (`newMsg`), fills it in, and unless the `newMsg` represents a completely filled board, creates a new chare for it. If it is a filled board, the solution is printed by the `printSolution` function. For printing data, one uses the call `CkPrintf` rather than `printf`. `CkPrintf` is atomic. I.e. data printed by a single `CkPrintf` is guaranteed to appear on the output without interference from other `CkPrintfs` being executed from other processors. To ensure that the solution is printed without being garbled by solutions coming from other chares, we collect the whole message to be printed in a buffer (using `sprintf`), and then print the buffer with `CkPrintf`.

Remember that when you receive a message (or when you allocate one), you own it, and when you send a message (or when you free it) you lose its ownership. So, the incoming message m is freed at the end of this chare's execution. The call `ChareExit()` frees up the resources used by this chare. It is also important to remember that when a message is sent, it should not be accessed in subsequent code. Thus, for example, it is incorrect to print the contents of the `newMsg` after the `CreateChare` call.

The execution of this program thus starts with creation of one instance of the `queens` chare. A `queens` chare typically fires many new chares, and so soon many chares run in parallel. Whenever any one of them finds a solution, it reports it via a `CkPrintf` call. Finally, (as the search space is finite) there are no more new chares to execute. Instead of letting processors wait indefinitely for new work, we would like to gracefully terminate the execution. This is accomplished by the `Quiescence` entry-point in the main chare. This is a special entry-point with a reserved name, and can be used only in the main chare. When there are no messages in the system, the runtime system sends a message to this entrypoint. After reporting time via the `CkTimer` call, we terminate the execution via the `CkExit` call.

```
module Nqueens {
#include <strings.h>
#define N 8
message { int nextRow; int Queens[N]; } PartialBoard;
message { int junk; } DUMMYMSG;
chare main {
  entry CharmInit:
     { ChareIDType mainid;
       PartialBoard * pMsg ;
       pMsg = (PartialBoard *) CkAllocMsg(PartialBoard);
       pMsg->nextRow = 0;
       CreateChare(queens, queens@NewBoard, pMsg);
       MyChareID(&mainid);
       StartQuiescence(Quiescence, &mainid); }
  entry Quiescence: (message DUMMYMSG *msg)
     {  CkPrintf("All Solutions reported. time = %d\n", CkTimer());  CkExit();  }
}
chare queens {
  /* No local variables */
  entry NewBoard: (message PartialBoard *m)
     { int col, i, row;
       PartialBoard * newMsg;
       row = m->nextRow;
       if (row == N) printSolution(m->Queens);
       else  for (col = 0; col < N; col++)
               if (consistent(m->Queens, row, col)) {
                   newMsg = (PartialBoard *) CkAllocMsg(PartialBoard);
                   newMsg->nextRow = row + 1;
                   for (i = 0; i < row; i++) newMsg->Queens[i] = m->Queens[i];
                   newMsg->Queens[row] = col;
                   CreateChare(queens, queens@NewBoard, newMsg);  }
       CkFreeMsg(m);
       ChareExit(); }
}
int consistent(queens, lastRow, col)
 int queens[], lastRow, col;
   { /* check if the new queen is safe from each of the previously placed queens */
     int x,y;
     for (x = 0; x < lastRow; x++)
       { y = queens[x];
         if ((y==col)  || ((lastRow-x) == (col-y)) || ((lastRow-x) == (y-col)) )
            return(0); }
     return(1);  }
printSolution(queens)
  int queens[];
   { int row;
     char buf[8*N], tmp[8*N];
     buf[0] = '\0';
     for (row = 0; row < N; row++)
       { sprintf(tmp, "(%d,%d) ", row, queens[row]);
         strcat(buf, tmp); }
     CkPrintf("%s\n", buf); }
} /* end module Nqueens */
```

3.1 Refining the simple Program

At this point we are ready to compile this program. Creating an executable parallel program in Charm requires translating the Charm program to produce a C file, compiling that to produce an object file and then linking it with various components of the run-time system on the desired parallel machine. In most cases this is done simply by copying a standard makefile and updating it slightly (in this case, just changing the name of the program). The standard makefile is shown below.

```
pgm : pgm.o
    charmlink pgm.o -o npgm

pgm.o : pgm.p
    charmc pgm.p
```

We change the name of the program from pgm to queens, assuming the program above was in the file queens.p, and run make. When I ran this program on my sparcstation, for N = 11, it created 166,926 chares and finished in 13.25 seconds (and printed 2,680 solutions!). The total number of chares created by the program can be obtained by using the `+ss` (for summary statistics) option while running the program. Thus the system created and processed about 13,000 chares per second! Although this says that the system overheads are low it is clear that we need to reduce the number of processes — and thus increase the amount of computation per process—if we are to get good efficiencies on parallel machines. How can we control this grain size? One simple solution is to use a preset threshold, say K. If the number of queens that are yet to be placed on the board is less than the threshold, we do the remaining work for this board sequentially without firing any new chares. As we are not sure what value of K will be adequate, we decide to experiment with different values of K. So, we want to read the value of K at runtime. Notice that the value of K gets set at the beginning of the computation and does not change thereafter. It is used by a large number of chare instances. The system does not allow unrestricted shared variables. Any sharing of information must be done through one of the six information sharing abstractions. Fortunately, readonly variables provide exactly the abstraction needed in this case. So we declare K as a readonly integer at the beginning of the module. The readonly variables must be initialized in the `CharmInit` entrypoint of the main chare, using the `ReadInit` function. (see Program 2, below).

Now we must write a sequential search function to be used below the threshold. Let us call it `sequentialQueens` . We must also add a test at the beginning of the `NewBoard` entry-point queens chare to test if the number of rows remaining to be filled is below the threshold and call `sequentialQueens` in that case.

One other thing. Let us make the program a little more flexible by making it work for any value of N (within reason) without having to recompile it. There are two problems here. As N is no longer a compile-time constant it must be treated as a shared variable and so should be implemented as a readonly variable. This involves changing all occurrences of N (except those in `CharmInit` entry in the main chare) by the expression `ReadValue(N)` . The second problem is that N is used to define the size of the queens array which is a part of the message. We will solve this problem simply, if somewhat crudely, by making the size of the array as large as the largest value of N we may want to use without recompiling. We will use the constant `Nmax` to denote this constant. The relevant portions of the new program is shown in program 2.

I compiled and ran this program on my workstation and tried different values of K for N = 11. The finish times and the number of processes created is summarized below.

```
grain=2: time = 9580 milli-seconds, chares= 164246
grain=3: time = 9060 milli-seconds, chares= 149130
grain=4: time = 7890 milli-seconds, chares= 114356
grain=5: time = 6460 milli-seconds, chares= 70208
grain=6: time = 5210 milli-seconds, chares= 32960
grain=7: time = 4550 milli-seconds, chares= 11598
grain=8: time = 4240 milli-seconds, chares= 3106
grain=9: time = 4170 milli-seconds, chares= 638
grain=10: time = 4110 milli-seconds, chares= 102
grain=11: time = 4120 milli-seconds, chares= 12
grain=12: time = 4110 milli-seconds, chares= 1
```

```
...
readonly int grain, N;

chare main {
  entry CharmInit:
  { CkPrintf("type N and grain-size:");
    CkScanf("%d %d", &N, &grain);
    ReadInit(N);
    ReadInit(grain); }
  ...
}

chare queens {
  /* No local variables */
  entry NewBoard: (message PartialBoard *m)
    { int col,i, row;
      PartialBoard * newMsg;
      row = m->nextRow;
      if (row == ReadValue(N)) printSolution(m->Queens);
      else if ( (ReadValue(N)-row) < ReadValue(grain))
        seqQueens(m->Queens, row);
      else
       ...
    }
}

seqQueens(queens, nextRow)
 int queens[], nextRow;
{ int col;
  if (nextRow == ReadValue(N)) { printSolution(queens); return; }
  for (col = 0; col<ReadValue(N); col++)
    if (consistent(queens, nextRow, col)) {
      queens[nextRow] = col;
      seqQueens(queens, nextRow+1);}
}
...
```

Program 2

As you can see, setting the threshold at 8 or 9 brings the time down almost to the minimum value. As a general rule we set thresholds so as to get smallest grain sizes that amortize (or mask) the overhead sufficiently. This leads to a large supply of processes to be dynamically load balanced by the system without undue overhead. Next I ran this program on a 20 processor Sequent Symmetry – a shared memory machine. All that was needed was to copy the program to this machine, use the standard makefile for Sequent and to compile the program again. I ran the program with N = 11 and K = 8, and it finished in 58.6 seconds. To run it with 10 processors I simply added (+p 10) to the command line, and it ran in 8.34 seconds. With two processors it ran in 29.7 seconds.

I noticed that it was producing a large number of solutions and therefore was spending much of its time printing them. For larger values of N, generating all the solutions seemed unnecessary. Instead I was interested in the number of solutions for larger values of N. How can we make the program count the solutions? In a sequential program I might have defined a variable, say count, and incremented it every time a solution was found. In the parallel program this will not be allowed as count would be a shared variable that needs to be updated from many different chares. Notice that the chares only add to the count, and never need to read it until the end of the computation. Thus count could be implemented as an *accumulator*, which is one of the information sharing abstractions supported by Charm.

The definition of the accumulator is shown below. Note that it only defines a new “type” (or class or data abstraction) called `AccIntPlus` .

```
message {} DUMMYMSG;
message {int value;} SINGLE_INT_MSG;
 ..
 ..
accumulator {

        SINGLE_INT_MSG *count;

        SINGLE_INT_MSG * initfn (data)
          DUMMYMSG *data;
        { count = (SINGLE_INT_MSG *) CkAllocMsg(MSG_LEVEL);
          count->value = 0;
          return(count);   }

        addfn (m)
          int m;
        { count->value += m; }

        combinefn (y)
          SINGLE_INT_MSG *y;
        { count->value += y->value;}

}  AccIntPlus;
```

An accumulator is defined with three functions named `initfn` , `addfn` , and `combinefn` . The data maintained by each accumulator must have the type of a message. In this case, `AccIntPlus` is defined with the message type `SINGLE_INT_MESSAGE`. The `initfn` is used to create the accumulator; It takes as its input a pointer to a message of some type - this type doesn't have to be the same as the accumulator's message type. This message is used for passing initialization data to the `initfn` . In this case, the initial value of the accumulator is 0, and thus no data is needed by the initfn. So, we define `DUMMYMSG` as a message with no fields, and define the parameter of `initfn` to be a pointer to it. The function allocates a message of the type matching the accumulator's message (in this case, `SINGLE_INT_MESSAGE`), fills it in with the initial value, and returns it. The returned message holds the "value" of the accumulator (i.e. it becomes "count" in this case). The addfn may take as many parameters as desired. Using these, it updates the current value in count. The combinefn tells the system how to combine two copies of the accumulator, if the system needs to. The user program never calls this function. It must take exactly one parameter, and its type must be a pointer to the accumulator message type. With this (and a bit of staring at the code), the accumulator definition should be evident.

To use `AccIntPlus` , we must create an instance of it by calling createAcc(..) system call. As it can be created in the `CharmInit` entrypoint, and as the instance ID is needed in many different chares, we define a readonly variable of type AccIdType, called solutionCount.

So, we add the above definition of `AccIntPlus` anywhere before the main chare, and add the following declaration anywhere before the main chare:

```
readonly AccIdType solutionCount;
```

In the `CharmInit` entrypoint, we add the following lines:

```
DUMMYMSG *m;
ChareIDType mainid;
...
m = (DUMMYMSG *) CkAllocMsg(DUMMYMSG);
solutionCount = CreateAcc(AccIntPlus, m);
MyChareID(&mainid);
StartQuiescence(Quiescence, &mainid);
```

Now, we are ready to add to the accumulator. We replace the call to `printSolution` by the call `Accumulate( ReadValue(solutionCount), addfn(1))`. [1]

We need to collect the final value of the accumulator. This collection should be invoked when we are sure there will be no more calls to `Accumulate` . So, the `quiescence` entrypoint offers a natural choice. We replace the code there with:

```
entry Quiescence: (message DUMMYMSG *msg)
  { ChareIdType mainid;
    mainchareId(&mainid);
    CollectValue(  ReadValue(solutionCount), main@RcvCount, mainid); }
```

This tells the system to send a message pointing to the final value of the accumulation solutionCount to main chare at the (new) entrypoint RcvCount.

We now declare the RcvCount entrypoint inside the main chare (anywhere in it).

```
entry RcvCount: (message SINGLE_INT_MSG *finalCount)
    { CkPrintf("The total number of solutions to %D queens is: %d\n",
                ReadValue(N), finalCount->value);}
```

[1] Those of you familiar with C++ may want to think of this as a syntactic variant of: `ReadValue(solutionCount)->addfn(1)` .

This completes the change to the program. (The particular accumulator used here is very commonly needed. It is therefore included in a system library module called `Accumulators` .)

I then ran this program on the Sequent Symmetry with 10 processors, and used N=11 and K=8 as input. The program reported finding 2680 solutions in 5.05 seconds.

Next, I ran this program on a NCUBE with 512 nodes. To do this, I copied the program from to a new directory on the ncube, copied the standard Makefile for ncube provided along with the distribution, changed the name of the file in the Makefile, and compiled it. (The directions for compiling and running programs on different machines are given in the manual). The program with N=11 and K=8 ran in 36.4 seconds on 1 processor, and generated 3,106 chares. The same program on 8 processors ran in 5.25 seconds. Going for larger problems, I ran it for N=14, k=8, on 64 processors, where it finished in 151.2 seconds.

What I'd do after this depends on who I am. A computer scientist would run this program for different number of processors, and plot nice speedup plots. An application scientist would run the program for as large values of N as feasible, and a computational scientist would aim at improving the algorithm (which is quite naive) so as to be able to solve larger values of N within the same amount of time!

4 The Jacobi Iteration With A Five-point Stencil

As our next example, we will consider a simplified version of a problem that occurs in numerical solutions for partial differential equations. This problem involves a 2-dimensional space. Each point in space has some attribute (say "pressure" or "potential"). The problem is to find the attributes at all points such that they satisfy a particular differential equation (called the Poisson equation). The value of the attribute at some of the boundaries is given. For the sake of concreteness: let the space under consideration be a unit square between co-ordinates (0,0) and (1,1).

For a numerical solution to this problem, one first discretizes it by dividing the space into a NxN grid. Now the problem is that of finding the values of the variable at each of the grid-points. The constraint that the value at a grid-point must satisfy the differential equation can now be (approximately) expressed as an equation between the difference of the value of the variable at neighboring points. Thus, there are now about NxN simultaneous equations to be solved.

One class of methods to solve linear equations such as these is that of iterative methods. The particular iterative method we will illustrate is called the Jacobi method. Here, one starts with some initial values (often just 0) for the variables except those fixed in the boundary by the input. Let array P hold the value of the variables. (such that P[x][y] has the value of the variable at grid-point (x/N,y/N)). In each iteration, the P array is updated in two steps. First, an intermediate array Q is computed such that P[i][j] is the average of its four neighboring points. Then P is replaced by Q. This process continues until the change in value of P anywhere in the grid is much smaller than the minimum value of P anywhere in the grid. This computation can expressed sequentially as shown in the the program on the next page.

```
Initialize(P); /* values in P */
repeat
     update(P, Q);  /* update Q using P */
     copyP(P, Q);   /* Copy Q into P */
     findMax(&maxDiff, &maxPrev, P, Q);
until  relError(maxDiff, maxPrev) < THRESHOLD
```

where relError is a macro, and initialize, update and copy are procedures defined as follows:

```
#define relError(x,y)   (y>0)?x/y:x;

initialize(P)
 float P[N][N];
  { int i,j;
    for (i=0; i<n+2; i++)
      for(j=0; j<n+2; j++) Q[i][j] = P[i][j] = 0.0;
        if (posi == 0) for(i=1; i<=n; i++) P[0][i] = 1.0;
 }

update(P, Q)
 float P[N][N], Q[N][N];
 { int i,j;
   for(i=1; i<=N; i++)
     for(j=1; j<=N; j++)
        Q[i][j] = 0.25*(P[i-1][j]+P[i+1][j]+P[i][j-1]+P[i][j+1]);
 }

copyP(P,Q)
 float P[N][N], Q[N][N];
 { int i,j;
   for(i=1; i<=N; i++)
     for(j=1; j<=N; j++)
        P[i][j] = Q[i][j];
 }

#define mymax(x,y) (x>y?x:y)
#define myabs(x) (x<0?-x:x)
private localMaximum (maxDiff,maxPrev,P,Q)
 float *maxDiff, *maxPrev;
 float P[N][N], Q[N][N];
  { int   i,j;
    *maxPrev = myabs(P[1][1]);
    *maxDiff = myabs(Q[1][1] - maxPrev);
    for(i=1; i<=n; i++)
      for(j=1; j<=n; j++) {
       *maxDiff= mymax(maxDiff, myabs(Q[i][j]-P[i][j]));
       *maxPrev= mymax(maxPrev, myabs(P[i][j]));
      }
  }
```

Program: Sequential Jacobi Algorithm

To parallelize this algorithm, we first note that each iteration of this algorithm is inherently parallel. Each element of Q can be computed independent of whether any other element of Q is computed or not. Also, the computation of the new value of Q[i][j] depends solely on the 4 neighboring values in the 4 primary directions - for ease of reference let us call them NORTH, EAST, WEST and SOUTH. At this point, we decide that the arrays will be divided in P equal parts (where P is the number of processors), and P identical processes - one on each processor - will each deal with updating their own portion of the array. Many different partitioning schemes can be employed, but we choose here a 2-dimensional (blocked) partitioning, where each process will own and maintain a nxn square of each array. (the number of processors, P must be a square for this to work without modification: i.e. P = pxp, for some p. Also, p*n = N). As all the processes are identical, and we need one on each processor, a branch-office chare is an appropriate construct to use here. Each branch must determine which section of the array it holds. We will use the following convention for this purpose: Let all the partitions be indexed by two co-ordinates. (So we have in effect a (N/n) x (N/n) array of partitions). Let the processors be numbered 0..P-1. If a processor is numbered `penum` , it holds a partition indexed by `(penum/p, penum % p)` where % denotes the modulus operator. Thus, with 64 processors, a processor with serial number 35 will own the partition indexed by (4, 3). If the total size of the array was 256x256, this processor owns the sections of P (and Q) P[32*4..32*5-1 , 32*3..32*4-1] (i.e. P[128..159, 96..127]).

As the computation of Q in the update phase requires the values of P at the four neighboring points, every processor will need to access the boundary values from the neighboring processors. The only way to access these values is for the neighboring processors to send them in messages. Therefore, in every iteration each processor sends its each of its borders to the appropriate neighboring processors, waits for messages from al its neighbors, and then carry out the update of the section of Q it owns. The update operation will be inconvenient and nonuniform if the boundary values found in the messages are stored separately. Instead, we allocate a slightly larger array ((n+2)x(n+2)) to accommodate the boundaries. The data in a message should then be copied into the appropriate part of the array when the message arrives.

We break down the computation of relative error into two parts. First, each processor computes the values of maxDiff and maxPrev based on the local sections of P and Q. Then, we must find the maximum of each variable across all the processors. The system library provides modules for several types of "reduction's". The module for the `maximum` function for a vector of `float` s is called `FMaxRedn` . The interface to this module (defined in `fmaxredn.int`) is via two functions: the `Create` function creates a BOC instance that will carry out the reduction. This function needs the size of the vector to be reduced as a parameter, and returns the ID of a BOC it created. Once created, the BOC can be used for multiple non-overlapping reductions that are of compatible type and size. The second function, `depositData` , takes this ID and the address of the base of the array of data to be reduced as parameters. It also takes the address another array (possibly same as the input array) to store the results of the reduction. In addition, it takes the name of a public function and an ID for a BOC (usually, the calling BOC). When the reduction is done, the reduction module calls this function of the specified BOC, informing that the reduction is now stored in the result array.

As all the processors execute essentially the same code on their part of the array, we choose to specify the computation as a BranchOffice chare called `Jacobi` . As the `Jacobi` code can be used in many different contexts, we will define it in a separate module. The main module would then simply start the reduction BOC, the Jacobi BOC, and wait for them to finish.

The main module (stored in a file, say main.p) is shown below.

```
#include "jacobi.int"
#include "fmaxredn.int"
module jacobi_example  {

message { int junk; } DUMMYMSG;

chare main  {
```

```
  entry CharmInit:
    { Jacobi::DOMAINInit *msg;
      ChareNumType       jacobi_boc;
      ChareIDType mainid;

      msg        = (Jacobi::DOMAINInit *) CkAllocMsg(Jacobi::DOMAINInit);
      msg->redn  = FMaxRedn::Create(2);
      jacobi_boc = CreateBoc(Jacobi::domain,Jacobi::domain@BranchInit,msg);
      MyChareID(&mainid);
      StartQuiescence(Quiescence, &mainid);
    }
  entry Quiescence: (message DUMMYMSG *msg)
    { CkExit(); }
 }
}
```

For this example, I have chosen to use a `quiescence` entry-point for detecting the termination. An entry-point named by the `StartQuiescence` call is enabled to be activated when there are no messages in the system and all the processes are idle. (Instead of `quiescence` , one may use many other ways of terminating the program. E.g. Each process may inform a `barrier` reduction (defined in the reduction library) when it is done, and the `barrier` will invoke an entry-point in the user program when all processors are at the barrier, which can then call `CkExit` (). Calling `CkExit` () directly in the `result` function is not a good idea, because although the computation may have finished on one processor, another processor may not have received the last reduction result yet. Calling `CkExit` () prematurely may stop such processors before printing their results, for example.)

The code for `Jacobi` is shown in the following pages. The `Jacobi` module imports the library module called `FMaxRedn` . In addition, its own interface file is included for checking consistency of the interface. There is no need to import the `main` module as `jacobi` does not use any entity defined in `main` .

The entry-points and functions on the first page are most relevant for understanding the parallel working of the program.

Each branch of the `Jacobi` BOC begins execution on its processor by executing the BranchInit entry-point. It first records the ID of the reduction BOC in a local variable, and frees up the message. It then initializes its portion of the P array, as well as some bookkeeping information such as the identity of the "neighboring" processors. It then sends its boundaries in separate messages to its (up to) 4 neighbors, and suspends. (As this is the first time through the iterations, there is no need to check for convergence. So, it sets the `reductionDone` flag.) The process is awakened when there is a message from one of its neighbors addressed to the `recvBoundary` entry-point. The code at this entry-point copies the boundary into the local array and frees up the message. If messages from all its neighbors have arrived, and the convergence test is done, it starts the next iteration by calling `iterate` , which is a function local (i.e. private) to this branch. `iterate` computes the `Q` array (via `update`), computes the maximum change and absolute value as in the sequential code (restricted to its own section of `Q` and `P`), and then deposits the `localMax` array with the reduction BOC. When the reduction is done, that BOC updates the `localMax` array, and calls the recvReduction function. If the relative error indicated by the values of the two maxima are below the threshold, `recvReduction` calls the `result` function, and suspends. Otherwise, it enables the next iteration by (copying the Q array into P,) sending the boundaries of P to neighbors and setting the `reductionDone` flag. The `if (count == 0) ..` statement is worth noting: It is possible that all of the neighbors received their reduction results and sent messages to this branch before it received its reduction results. Thus, it must check if all the neighbor messages have been received, and call `iterate` if so. Conversely, if the reduction results have arrived first, the `recvBoundary` entry-point checks for it, and calls `iterate` after all neighbor messages are received.

```
#include "jacobi.int"
#include "fmaxredn.int"
module Jacobi {
#define n          10   /* nxn is the size of the sub-domain */
#define THRESHOLD        1e-4
#define EAST  0
#define WEST  1
#define NORTH 2
#define SOUTH 3
#define TRUE  1
#define FALSE 0
message { ChareNumType redn;}  DOMAINInit;
message { int from;  float boundary[n+2]; }  BOUNDARY;

BranchOffice domain {
       float P[n+2][n+2],Q[n+2][n+2];
       float localMax[2];
       int   count,numOfNeighbours;
       int   reductionDone;
       int   valid[4], penum;
       int   K;                          /* KxK is the number of processors */
       ChareNumType redn, mybocid;

       entry BranchInit: (message DOMAINInit *msg)
         {   redn = msg->redn;
             CkFreeMsg(msg);
             mybocid = MyBocNum();
             PrivateCall(initialize());
             PrivateCall(sendBoundaries());
             count = numOfNeighbours;
             reductionDone = TRUE;  }

       entry recvBoundary : (message BOUNDARY *msg)
         {   private void iterate();
             PrivateCall(copyBoundary(msg->from,msg->boundary));
             CkFreeMsg(msg);
             if (--count == 0 && reductionDone) PrivateCall(iterate());  }

       private void iterate()
         {   update(P,Q);
             localMaximum( &(localMax[0]), &(localMax[1]),P,Q);
             reductionDone = FALSE;
             count = numOfNeighbours;
             FMaxRedn::DepositData(redn, localMax, localMax, recvReduction, &mybocid);  }

#define relError(x,y) ((y>0) ? x/y : x)
       public recvReduction()
         { if (relError(localMax[0],localMax[1])  < THRESHOLD )
                result(penum,K,P);
           else { reductionDone = TRUE;
                  copyP(P,Q);
                  PrivateCall(sendBoundaries());
                  if (count == 0) PrivateCall(iterate()); } }
```

```
    private initialize()
      {   int i,j,posi,posj, Pe;
          extern double sqrt();
          penum = McMyPeNum();
          Pe = McMaxPeNum(); K  = (int) sqrt((double) Pe);
          if (K*K != Pe) { CkPrintf("error. The number of processors must be a square.\n");
                           CkExit();}
          posi  = penum / K;   posj  = penum % K;
          numOfNeighbours = 4;
          for(i=0;i<4; i++) valid[i] = TRUE;
          if (posi == 0)   {numOfNeighbours--; valid[NORTH] = FALSE;}
          if (posi == K-1) {numOfNeighbours--; valid[SOUTH] = FALSE;}
          if (posj == 0)   {numOfNeighbours--; valid[WEST]= FALSE;}
          if (posj == K-1) {numOfNeighbours--; valid[EAST]= FALSE;}
          /* initialize external boundaries */
          for (i=0; i<n+2; i++)
              for(j=0; j<n+2; j++) Q[i][j] = P[i][j] = 0.0;
          if (posi == 0) for(i=1; i<=n; i++) P[0][i] = 1.0;
      }

    private sendBoundaries()
      { int i,dir,destPe;
        BOUNDARY *msg;
        for(dir=0; dir<4; dir++)
           if (valid[dir]) {
              msg = (BOUNDARY *) CkAllocMsg(BOUNDARY);
              switch (dir) {
                 case NORTH : msg->from = SOUTH;  destPe = penum-K;
                              for(i=1; i<=n; i++) msg->boundary[i]=P[1][i];
                              break;
                 case SOUTH : msg->from = NORTH;  destPe = penum+K;
                              for(i=1; i<=n; i++) msg->boundary[i]=P[n][i];
                              break;
                 case EAST  : msg->from = WEST;   destPe = penum+1;
                              for(i=1; i<=n; i++) msg->boundary[i]=P[i][n];
                              break;
                 case WEST  : msg->from = EAST;   destPe = penum-1;
                              for(i=1; i<=n; i++) msg->boundary[i]=P[i][1];     }
              SendMsgBranch(recvBoundary,msg,destPe);
           }
      }
    private copyBoundary(from,boundary)
    int    from;
    float  boundary[];
      {   int i;
          switch (from) {
            case NORTH: for(i=1; i<=n; i++) P[0][i]    = boundary[i]; break;
            case SOUTH: for(i=1; i<=n; i++) P[n+1][i] = boundary[i]; break;
            case EAST : for(i=1; i<=n; i++) P[i][n+1] = boundary[i]; break;
            case WEST : for(i=1; i<=n; i++) P[i][0]    = boundary[i];
          }
      }
}
```

```
        update(P,Q)
        float P[n+2][n+2], Q[n+2][n+2];
          { int i,j;
            for(i=1; i<=n; i++)
              for(j=1; j<=n; j++)
                Q[i][j] = 0.25*(P[i-1][j]+P[i+1][j]+P[i][j-1]+P[i][j+1]);
          }

        copyP(P,Q)
        float P[n+2][n+2], Q[n+2][n+2];
          { int i,j;
            for(i=1; i<=n; i++)
              for(j=1; j<=n; j++) P[i][j] = Q[i][j];
          }

#define mymax(x,y) ((x>y)?x:y)  /* if (x>y) then x  else y  */
#define myabs(x) ((x<0)?-x:x)   /* if (x<0) then -x else x */
        localMaximum (maxDiff,maxPrev,P,Q)
        float *maxDiff, *maxPrev, P[n+2][n+2], Q[n+2][n+2];
          { int   i,j;
            *maxPrev = myabs(P[1][1]);
            *maxDiff = myabs(Q[1][1] - *maxPrev);
            for(i=1; i<=n; i++)
              for(j=1; j<=n; j++) {
               *maxDiff= mymax(*maxDiff, myabs((Q[i][j]-P[i][j])));
               *maxPrev= mymax(*maxPrev, myabs(P[i][j]));
              }
          }

        result(penum,K,P)
        int penum,K;
        float P[n+2][n+2];
          {  int i,j;
             for(i=1; i<=n; i++)
               for(j=0; j<=n; j++)
                 CkPrintf("(%d,%d) %f\n", (penum/K)*n+i, (penum%K)*n+j, P[i][j]);
          }
}
```

The procedures `initialize`, `sendBoundaries` and `copyBoundary` carry out the details of neighborhood communication, and are fairly straightforward. (Also, they are essentially the same procedures as those needed when this algorithm is expressed in most of the other explicitly parallel languages/systems. The last set of procedures, `update`, `copyP`, `localMaximum`, and `result` are almost identical to their sequential counterparts.

The first set of entry-points and procedures, including `BranchInit`, `recvBoundary`, `iterate`, and `recvReduction` express the essence of the parallel algorithm in Charm. Note that after sending the boundaries to the neighbors, as well as after requesting the reduction, the chare simply suspends, to be awakened when the relevant message/s arrive. In the meanwhile, other parts of user computation, if any, can proceed with their execution. This is one of the major benefits of message-driven execution as supported in Charm. The need to maintain the `count` and the `reductionDone` flag is the price paid for this. (However, a new tool, called Dagger, can eliminate the need for such flags in most cases.)

5 A Prime Finder

Consider the following problem: find and store all the prime numbers between 2 and N, for a given N. For concreteness, and to clarify some design choices, assume that we are aiming at N of the order of about a billion.

We will examine the process of design of this algorithm, illustrating various design choices one has to make in developing a parallel program.

First, we must choose a parallel algorithm. For this problem, we will first choose a sequential algorithm, and then decide how to parallelize it. (The alternative is: choose a parallel algorithm in the first place.) One obvious algorithm is: test each number i between 2 and N for primality. We can ignore the even numbers. Also, to check if i is prime, it is sufficient to test if each prime p, $p^2 \leq i$, does not divide i evenly. One can start testing i by dividing by the smallest prime (3, assuming we are not considering even numbers), and continue testing i only as long as we haven't found an even divisor. Starting at the smaller end is better because it is more likely that a given number (i) is divisible by a smaller prime than by a larger one, and thus we may rule it out sooner. The sequential algorithm is shown below:

```
primes[0] = 2;
nextPrimeAt = 1;
for (i=3; i<N; i += 2)
     { isPrime = TRUE;   j = 1;
       p=primes[0];
       while ( isPrime  && (p*p <= i))
         { if ( (i % p) == 0)
             isPrime = FALSE;
           else p = primes[j++];}
       if (isPrime) primes[nextPrimeAt++] = i;
     }
```

However, the Eratosthenis' Sieve algorithm, described below, is more efficient than the above algorithm. This algorithm starts by initializing a vector of length N to all 0's. The value at i'th index corresponds to the status (potentially prime or non-prime) of the number i. A 0 value at index i indicates that i is potential prime, but a 1 indicates that i has been shown to be a composite (i.e. non-prime). We will ignore the index 1 in the following discussion. The algorithm proceeds as follows: in each step, the next prime is found by looking for the smallest unexamined 0. Suppose that is found at index p. All the multiples of p in the vector are then marked as composites by setting the corresponding values to 1. It is clear that this procedure needs to continue only until p passes sqrt(N): if a number i, $i \leq N$ is a composite, then surely it must have a prime divisor q that is not larger than sqrt(N). Also, as before, we may omit even numbers from consideration by using a vector of size N/2, and using the convention that the i'th value represents the number 2*i+1. Conversely: the number k is represented by the bit at index (k-1)/2 However, for ease of presentation, we will postpone that as well as other possible optimizations.

The sequential algorithm is shown below:

```
for (i=1; i<=N; i++) vector[i] = 0;
nextPrime = 2;
while (nextPrime*nextPrime <= N) /* Mark multiples of nextPrime */
 { for (multiple = 2*nextPrime; multiple <= N; multiple += nextPrime)
       vector[multiple] = 1;
   nextPrime++;
   while (vector[nextPrime] == 1) nextPrime++; /* find the next Prime */
 }
```

It is useful to estimate the sequential completion time for this algorithm: The inner loop is executed once for each prime not larger than $\sqrt{N}$. The number of iterations of the inner loop depend on the value of the prime. For a prime p, the inner loop body will be executed ceiling(N/p) times - as p can be at most sqrt(N), this approximates closely to N/p. Thus, the overall time can be expressed as a sum: $\sum N/p$ over all primes $p \leq \sqrt{N}$. I.e. N*(1/2 + 1/3 + 1/5 + 1/7 ... 1/x) where x is the largest prime not larger than $\sqrt{N}$. Empirically, this sum remains less than 2.2*N for N up to a million. (Incidentally, this was found out by using the above code, sequentially, to find all the primes up to 1000, and then computing the above sum. Such "probe computations" are often very important in making design choices).

One immediate consequence of this time-complexity estimate is the realization that I/O cost is likely to be comparable to (or even worse than) the computation cost - both are O(N). (Remember that the vector must be stored in the file system, as a part of the specification.) Therefore, we must consider concurrent I/O, assuming it is supported by the machine we are running the program on. We will return to this point later.

Having selected the sequential algorithm, let us decide how to parallelize it. A cursory inspection of the code suggests two possibilities: we can partition the vector in multiple parts and work on producing different partitions in parallel. Alternatively, we can mark multiples of different primes in parallel. (The former corresponds to parallelizing the inner loop, whereas the latter corresponds to parallelizing the outer loop). As a third alternative, we can use both sources of parallelism together. Choosing between these three alternatives is the first design decision we have to make. Let us consider them in some detail.

The first alternative: We must somehow split the range 1..N into many partitions, each of some length g. We have to decide whether to use static load balancing or dynamic load balancing. As the work involved in each section of the vector may vary, we choose (somewhat arbitrarily) to use dynamic load balancing, and hope to create a large number of partitions so that dynamic load balancing strategy provided by Charm will do a better job of distributing the work evenly. The choice of the exact value for g can be postponed until after we have finalized the parallelization technique. To split the range, one may use many methods. For example:

1. a We may have the main chare create all the chares, one for each partition, directly. However, the main chare may become a sequential bottleneck in that case, having to process all the (N/g) createChare requests.
2. b The main chare creates a chare C for the range 1..N. C first fires a chare (of type C again) for the range g+1..N, and works on the first partition. Other instances of g, likewise, work on the leftmost partition, after firing a chare for the remaining range. The problem with this method is that only one chare is available for distribution at any time, which is not good for load balancing, and certainly not a solution that will scale up to large distributed memory machines.
3. c Our method of choice will be divide and conquer- each chare receive a range, L..U. If it is small enough (no larger than g) then the chare works on the bitvector directly. Otherwise, it splits the range in two roughly equal parts, and fires off two chares, one for each range.

The second alternative: Two chares mark the vector using different sets of primes. As we cannot have such writable shared data structures in Charm programs, we must have them work on two copies of the vector, and then combine the two by performing a logical OR. Also, as marking all multiples of smaller primes is more work than those of larger primes, we cannot simply evenly divide the primes. Finally, as the bitvector is large we have to split it in sections anyhow so as not to exceed the available storage.

The second alternative involves sending additional messages to parents (containing the bitvector) compared to the first alternative. Also, it involves the additional work of performing the logical OR operation. Because of these factors, combined with the fact that we can always split the sub-range if we need additional parallelism, it is clear that the first alternative should be chosen. Since the first alternative clearly dominates over the second, the third alternative that combines the first two is not worth pursuing. If we want to split our subcomputations further, it is always beneficial to divide up the sub-range rather than split the range of marker primes.

The parallel action in the above program is to compute the primes within a sub-range L..U in a vector. However, we notice that the vector itself is a shared data structure. To compute a section of the vector, we need access to earlier portions of the bit-vector to find the set of primes with which we need to mark the given section. This data-dependency must be dealt with. We must decide what specific information sharing mode we should use for accessing the vector. As each partition is computed, we may store it in a distributed table with a suitable index. Chares that need a particular partition can fetch it from this table, using the table operation "find". However, we notice another empirical fact: only a small initial section of the vector is really shared - that representing numbers between 1 and $\sqrt{N}$. For N = 1 billion, this is roughly 32,000 (actually: 31,623). To mark multiples of primes between 1 and 32,000, it may be simplest to compute these primes at the beginning of the computation, and store them as *readOnly* variables.In doing this, we are creating an inherently sequential component (of computing and storing all the primes in the range 1..32,000, costing about 32,000*2.2 operations by our estimate above). However, compared to the rest of the computation (about 2.2 billion bit-setting operations, and storing of the resultant primes), this is a small cost, and won't be a significant until we start using many thousands of processors. If it had turned out to be a significant factor: we would simply compute this list of primes up to 32,000 using a parallel method, and then establish it as a writeOnce variable.

Once a chare computes the primes between a subrange L..U, they must be stored. As we stated earlier we must use a concurrent output scheme for this. Given P processors, we choose to have each processor i (i between 0 and P-1, inclusive) store the primes in the range $(N * i)/P..(N * (i + 1)/P - 1$. Thus, there will be P separate output files generated by this program. To distinguish them, we will name them primes0, primes1, ... The subranges L..U, however, are being distributed using dynamic load balancing, and so may be computed on any of the P processors. So, we decide that each chare, when it finishes computing a section of the bitvector, turns it over to a branch-office chare, which handles its storage. This BOC, when called via the branch function *recordPrimes* will first calculate which processor the data should be stored by, and then send it to the branch on the desired processor. A branch that receives bitvector from another processor will buffer it until it can be stored on a file. Note that dynamic load balancing and the inherent asynchrony in the order in which work is picked up implies that the sections of the bitVectors may be computed in arbitrary order. The output file must have the records for each section appear in sequence. This is the reason why sections of the vector may have to be buffered.

At this point, we have also implicitly made a decision to store the output file as a sequence of records, where each record is a different section of the bitVector. For clarity, we assume that each record will begin with two numbers: the starting integer, and the length of the record. We will also modify the storage rule above slightly to simplify the storage decision (and to avoid having to split a bitVector computed by a chare). If the first number of a section of BitVector (i.e. if the bitvector represents the range L..U, then L is the first number) is between $i * N/P..(i + 1) * N/P - 1$, the section is stored at processor i.

The program that results from these decisions at this point is shown on the last 2 pages, in pseudo-code form.

Optimizations

Before we run it "full-strength" (i.e. on the range 1..1 billion), we first try to optimize it. This can be facilitated by running it on small ranges and generating some profiling information. In this case, we will simply use observations about this program.

The memory used by the program is dominated by the buffering needed in the branch-office chares. The buffering requirement is smaller if earlier sub-ranges *in each processor's domain* are computed and returned to it earlier. For example, if a processor is responsible for storing primes between 10 million and 12 million, it is better to get sections of bitVector closer to 10 million before those closer to 12 million. Another processor, storing sections between 12 and 14 million will prefer to have sections in that range that are closer to 12 million generated before those closer to 14 million. This can be facilitated by assigning a priority to each section L..U. (Exercise: figure out the priority to be attached to each createChare message).

The most profitable optimizations are usually those applied to sequential portions of the code. In this program, the sequential code (in sieve chare) that marks the bits in a sub-range (L..L+g) dominates the computation time. Multiples of all primes up to the square root of (L+g) must be marked. As before, this costs about g(1/2+1/3+1/5+1/7...) operations. It is clear that it takes more time to mark multiples of smaller primes. We can avoid marking multiples of 2 by simply storing only the odd numbers in the bitVector (so i'th bit represents i'th odd number: 2*i+1). Is it possible to avoid marking multiples of 3 (and other small primes)? (Think about this before you read on).

If we choose the grain-size g to be 2*3*5*7 = 210, and make sure that the divide-and-conquer is done in such a way that every subrange (L..U) starts at a multiple of g (i.e. L = k*g), we can employ the following technique: A bitvector of size 210 (actually 115, if we omit even numbers) can be initialized so that multiples of 3, 5 and 7 are marked in it. This takes roughly 210*(1/3+1/5+1/7) = 142 bitVector operations. Now we can make this initialized bitVector "template" available as a readOnly variable. Every time the sieve chare starts to create a bitVector for the range L..U, it now initializes it to this template, and then starts marking multiples of 11 and higher primes. (Why does this work? Remember L is a multiple of 210. So, the pattern of bits to be set as multiples of 3,5 and 7 are identical to the template). The initialization involves copying 115/32, or roughly 4 words, compared to 142 operations saved. In fact even this copying cost is not an "extra", as we would have to initialize the vector to all 0's to begin with, anyway.

The only problem with the scheme above is that the grain-size is too small. Each leaf of the divide-and-conquer tree (of the sieve chare) accounts for two messages for creating it (1 for its actual creation message, and one for its share of the creation messages for the internal nodes of the creation tree.) In addition, it results in sending one message to the processor where it should be stored (except in the lucky situation when that happens to be the same processor where it was computed). We must have a few thousand operations per message to amortize the message overhead.

This is easy to fix. Instead of going up to 7, we go up to 13, and get g=30,030. As reasoned above, the work involved in marking a vector for the range L..L+30030 (omitting even numbers) is roughly: 15015*(1/17+1/19+1/23.. 1/x) where x is the largest prime smaller than square-root of N. This seems like a reasonable range to try. (The next possibility is g = 30030*17 = 510,510 in which case the bitVector section representing half a million numbers will likely be too large and cause load imbalances. In any case, we can check how the grain-size works out on small probe computations, and go to the large size if needed.)

With this optimization, we now have a complete and efficient prime finder. After some probe computations (finding primes up to a million, say), we decide that the grain-size is adequate. (How? by finding out the sequential execution time using the sequential algorithm above, and then dividing by the number of messages generated in the parallel algorithm, we get the average grain-size). We also use the probe computations to ascertain that our algorithm is producing the correct list of primes. It would be nice if we could prove that. Stopping short of that, we could check its output against a correct algorithm for relatively small values of N. We also run a few checks with N = 1 million (say) by running the program with one and more processors, and checking if we are getting good speedups. Once we are reasonably sure that the algorithm will be effective for N= 1 billion, we run it on the largest configuration available to us (say an NCUBE with 512 nodes), to finally store the list of primes in that range.

This program, when run on a 32 node iPSC/860, was able to compute (and count) all the primes in the range 1..1 billion in 75 seconds. (Without storing them–storage was turned off because of the lack of disk space. Instead of storing the primes, the version I ran counted the numbers of primes in the given range.)

```
module main {
  ... Declaration of some read-only variables ...

message {int L; int H;}  goalMsg;
message {int Start; int length; int vector[(LENGTH)/64 + 1];}  RespMsg;
message { } DONEMSG;
message { int junk; } DUMMYMSG;

chare main {
        int M;
        ... Other local variable declarations...
entry CharmInit:
        { goalMsg * message;
          ChareIDType mainid;
          printf("Primes upto:");  scanf("%d", &M);
          ReadInit(Limit, M);
          ... Initialize the other read-only variables needed.
          ... create an instance of the BOC recordData
          msg = (goalMsg *) CkAllocMsg(sizeof(goalMsg));
          msg->L = 1;  msg->H = M;
          CreateChare(sieve, sieve@Goal, msg);
          MyChareID(&mainid);
          StartQuiescence(Quiescence, &mainid);  }
entry Quiescence: (message DUMMYMSG *msg)
        { CkExit(); }
}

chare sieve
{int count;
 entry Goal: (message goalMsg * msg1)
        { goalMsg * msg2;
          int Mid, L, H;
          L = msg1->L; H = msg1->H;
          if ((H-L+1) > LENGTH)  /* above the grain-size: divide */
           {  Mid = (H+L)/2;
              msg2 = (goalMsg *) CkAllocMsg( sizeof(goalMsg));
              msg2->L = Mid;   msg2->H = H;
              msg1->H = Mid-1; /* msg1->L == L already */
              CreateChare(sieve, sieve@Goal, msg2);
              CreateChare(sieve, sieve@Goal, msg1); }
          else {bitVector = seqSieve(L,H); /* computes the primes in the
                                        range L-H in the bitVector */
                BranchCall(recordBocNum, recordPrimes(bitVector, L, length));}
        }
}
```

```
BranchOffice recordData {
     int storedUpto;
     RespMsg  ** arrayMsgs;
     int size;
     int firstStart;
     FILE * fd ;
   entry BranchInit: (message RecordCreationMsg *msg)
        { .... Allocate and initialize arrayMsgs to buffer the messages.
          storedUpto = 0; /* nothing stored on file yet. */
          firstStart = (msg->limit * McMyPeNum()) / McMaxPeNum();
          ... create a filename by concatenating pe number to the string \"output\"
          .. then open the file for output, obtaining its descriptor in fd.}
    recordPrimes(vector, startingFrom, length)
        int * vector;
        int startingFrom, length;
        { int peNum, i;
          ... Copy the vector into a message msg.
          peNum = (McMaxPeNum() * startingFrom) / ReadValue(Limit);
          SendMsgBranch(ReceivePrimes, msg, peNum);}
    entry ReceivePrimes: (message RespMsg *msg)
        {  int index, i ;
          /*  put them in a file, if you can. Buffer them otherwise */
           index = (msg->Start - firstStart)/LENGTH;
           arrayMsgs[index] = msg;
          while ((storedUpto < size) && (arrayMsgs[storedUpto] != NULL)) {
            ...Write the record at this index in the file opened. The record
            ...consists of an int (startingfrom) followed by the bitvector.
            ...Free the buffered message.
            storedUpto++;}
        ...close the file fd if all the records have been received.
        }
  }
}
```

6 Discussion

All the programs discussed in the tutorial are also available on-line along with the Charm distribution. They can be found in a directory named *charm_pgms*.

The set of features covered by the tutorial examples is by no means exhaustive. The reader is encouraged to explore the Charm manual and the other tools included in the Charm distribution.