High-productivity software development with pMatlab

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>As Published</td>
<td><a href="http://dx.doi.org/10.1109/MCSE.2009.9">http://dx.doi.org/10.1109/MCSE.2009.9</a></td>
</tr>
<tr>
<td>Publisher</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>Version</td>
<td>Final published version</td>
</tr>
<tr>
<td>Accessed</td>
<td>Tue Dec 18 18:33:47 EST 2018</td>
</tr>
<tr>
<td>Citable Link</td>
<td><a href="http://hdl.handle.net/1721.1/60014">http://hdl.handle.net/1721.1/60014</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>Article is made available in accordance with the publisher’s policy and may be subject to US copyright law. Please refer to the publisher’s site for terms of use.</td>
</tr>
<tr>
<td>Detailed Terms</td>
<td></td>
</tr>
</tbody>
</table>
Although high-performance computing (HPC) has been around for decades, most working engineers and scientists don’t use parallel or distributed systems for their research or design because of the hurdles associated with using such resources. Matlab, a widely used programming tool for scientists and engineers, has the characteristics that appeal to the scientific community—namely, ease of use, rapid prototyping capability, and an interactive development environment. Bringing this segment of the scientific community to HPC requires a similar set of tools and features that abstract the hardware and parallel coding details away from the design of scientific applications.

Recognizing the need for such a library, the MIT Lincoln Laboratory created an interactive parallel development environment that includes pMatlab, an open source parallel Matlab library; MatlabMPI, an open source Matlab-compatible library for interprocessor communication; and gridMatlab, a proprietary library that transparently connects the user’s desktop Matlab session with \( N - 1 \) remote Matlab sessions on a large distributed cluster. In this installment of Scientific Programming, we explore the ease of tackling a communication-intensive parallel computing task—namely, the 2D fast Fourier transform (FFT). We start with a simple serial Matlab code, explore in detail a 1D parallel FFT, and illustrate how it can be extended to multidimensional FFTs.

**Parallel Development Environment**

Figure 1 shows the system’s general structure. The scientific programmer works at the application layer, while pMatlab abstracts the hardware architecture and system details. The library level abstracts the (parallel) data and task distribution through maps. The kernel layer is composed of math (such as Matlab or Octave), communication (such as MatlabMPI, bcMPI, or MPIToolBox), and cluster launch kernels. The cluster launch kernels control interactions between the desktop and remote Matlab sessions. Although our cluster launch kernel, gridMatlab, is proprietary, pMatlab can run interactively on any cluster.

Programming in pMatlab

The programming paradigm is best understood by example. To illustrate the commonly used pMatlab functions, we explore an example from signal and image processing. A common procedure in this domain is to read in data, process it via FFTs, send the results to another stage, or collect them for graphing or display. For simplicity, we show 2D arrays in the figure, but pMatlab supports distributed arrays for up to four dimensions. Creating the map requires the user to specify the array dimensions to be distributed, how it’s to be distributed, and the set of processors over which it’s distributed. Changing one of these parameters changes the data distribution. The mappings in Figure 2 clearly indicate that the map abstraction provides an easy way to evaluate a range of distribution strategies for a given application.

pMatlab, a parallel Matlab library, lowers the barrier to development of parallel and distributed application codes by providing an easy-to-use programming environment.
It’s less clear how to efficiently gather the data for display, or even how to approach processing a multidimensional FFT that requires distributing rows first and then redistributing the data along columns. In traditional parallel computing, this requires hand coding of the distribution and communication via the use of a message-passing interface (MPI) library, generally the open source MPICH or an architecture-tuned version. For the simple case of the 1D FFT, this method of coding requires that the programmer determine and specify where to send each row of data (which processor), who will collect the results (leader), and how each processor will send results to the leader. In the multidimensional case, the programmer is required to distribute the data for the row FFTs and then redistribute the data for the column FFTs, as well as collect the results. Although it’s easy to think about the 1D data distribution, the implementation details quickly become complicated, and additional dimensions increase the complexity. (The left-hand block of code in Figure 4 illustrates a Matlab-MPI program to perform a 2D FFT and provides a sense of the complexity associated with writing the data distribution and local-global mapping code.)

By contrast, to create a pMatlab version of the code as in Figure 5, we map the (independent) rows of data across processors by calling the map function as shown in line 7. Once we create the map, we can initialize the distributed array, DZ, by passing a map object to the zeros constructor. This creates a distributed matrix of zeros. Notionally, the dmat looks like the colored matrix in the right-hand side of Figure 5. Note that each color represents a separate processor.

The next step in the application is the assignment of the data to the distributed matrix, which we perform via the subsassign function (line 11). Each processor will perform an FFT on a subset of the total number of rows, and all processors execute the same instructions. The execution of the function local returns a copy of the local portion of the distributed matrix data so that processing can begin. Through map construction, each processor implicitly knows its data partition and the associated indices, but often the programmer needs these values to perform some other task that directly references the global indices. To provide this information, the pMatlab function global_ind returns a vector of global indices (for example, row numbers) for the associated rank (processor identifier). Once a processor has the local copy of the data, it executes the Matlab FFT function on the local data, myRows. When the processing is complete, the put_local function returns the data to the dmat object. Note that at
my_rank=MPI_Comm_rank(comm);
if (my_rank==0)|(my_rank==1)|(my_rank==2)|(my_rank==3)
Xlocal=rand(M,N/4);end
if (my_rank==4)|(my_rank==5)|(my_rank==6)|(my_rank==7)
Zlocal=zeros(M/4,N);end
Xlocal=fft(Xlocal);
tag=0;
if (my_rank==0)|(my_rank==1)|(my_rank==2)|(my_rank==3)
start=1;
len=M/4;
for dest_rank=4:7
last=start+len-1;
MPI_Send(dest_rank,tag,comm,Xlocal(start:last,:));
start=last+1;
end
end
if (my_rank==4)|(my_rank==5)|(my_rank==6)|(my_rank==7)
start=1;
len=N/4;
for recv_rank=0:3
last=start+len-1;
Zlocal(:,start:last)=MPI_Recv(recv_rank,tag,comm);
start=last+1;
end
end
Zlocal=fft(Zlocal);

X = fft(X,[],2);
Z(:,:) = X;
Z = fft(Z);
Z = transpose_grid(X);
Z = fft(Z);

Figure 4. Parallel two-dimensional fast Fourier transform (FFT) code. A comparison of MatlabMPI, pMatlab, and optimized pMatlab implementations shows the level of complexity that’s abstracted away from the programmer through the pMatlab library.

%Initialize pMatlab
pMatlab_Init;
Ncpus = pMatlab.comm_size;

%read input data
inData = readInputData;

% Create Maps – distribute rows
map1 = map([Ncpus 1],{},0:Ncpus-1);

% Create DZ – distributed matrix.
DZ = zeros([n, m, map1]);

Assign data to distributed array
DZ(:, :) = inData;

% Get the local portion and local indices
myRowNumbers = global_ind(DZ, 1);
myRows = local(DZ);

%perform FFT on rows
myRows = fft(myRows, [], 1);

% Copy local portion to global
DZ = put_local(DZ, myRows);

%Gather results to leader processor for display or next processing stage – Note: Result is a plain MATLAB® matrix.
results = agg(DZ);
pMatlab_Finalize;

Figure 5. pMatlab version. On the left, we see code for the pMatlab fast Fourier transforms, and on the right, the annotated distributed array.
% RUN is a generic script for running pMatlab scripts.
% Define number of processors to use
Ncpus = 4;
% Name of the script you want to run
mFile = 'param_sweep_parallel';
% Define cpus
% Run on user's local machine
% cpus = {};
% Specify which machines to run on
cpus = {'node-1', 'node-2', 'node-3', 'node-4'};
% Abort left over jobs
MPI_Abort;
pause(2.0);
% Delete left over MPI directory
MatMPI_Delete_all;
pause(2.0);
% Define global variables
global pMATLAB;
% Run the script.
[‘Running ‘ mFile ‘ on ‘ num2str(Ncpus) ‘ cpus’]
eval(MPI_Run(mFile, Ncpus, cpus));

Figure 6. Two-dimensional fast Fourier transform performance. Compare the results for three different approaches, MatlabMPI, pMatlab, and pMatlab using the transposegrid function.

Figure 7. pMatlab launch script. For a general distributed cluster, we set the number of processors on which to run, their names, and the names of the m-file and the machines on which the job is to be run.

this point, each processor only has the FFT results of the local data. The data on other processors needs to be communicated to this processor if required for a subsequent stage of computation or display. To aggregate the data for display or the next stage in a processing chain, the agg command will gather the data from all the processors. During this function’s execution, each processor sends its local data to the leader processor (0), which collects them in rank order. The result of the agg command is a plain Matlab matrix on processor 0 (the leader). By design, processor 0 is the user’s local desktop—the code has been running interactively in a Matlab session on the desktop and n − 1 processors in a remote system. Having completed the parallel processing, we close the remote processes using pMatlab_Finalize, leaving the result in the workspace of processor 0. In this way, we’ve maintained the interactive Matlab environment, but the user has achieved significant acceleration of his or her workflow. In general, such applications approach linear speedup, although collection in the agg function is inherently serial and thus a bottleneck.

pMatlab Code
This simple code we just described illustrates the required initialization and finalization functions, pMatlab_Init and pMatlab_Finalize, as well as the most commonly used functions, map, global_ind, local, put_local, agg, and the method of distributed matrix creation. We find that these are often the only functions required for signal- and image-processing applications in which the images fit in a single processor’s memory.

However, in many applications, the data doesn’t fit into the memory of a single processor, so distributed tensor products, matrix inverses, or multidimensional FFTs must be supported. The multidimensional FFT leads to the most communication-intensive operation in parallel computing, the corner turn or complete exchange (an all-to-all communication between processors). Building on our earlier example, let’s consider a 2D FFT code segment. The code, shown in Figure 4, is very similar to that in Figure 5;
the initialization steps are the same, but now there are two maps, one for the row-wise FFT and one for the column FFT.

Here, we illustrate three approaches to creating the parallel Matlab code: using MatlabMPI, which requires the programmer to compute the assignment by hand, using pMatlab to remap the data distribution between the row and column distributions, and using a specialized pMatlab function, transpose_grid, to optimally remap the data between the two FFT stages. We include the MatlabMPI code to provide some sense of the level of complexity involved in the corner turn or remapping. Figure 6 plots the speedup achieved in each approach. Clearly, the optimized pMatlab version compares well with the MatlabMPI results for significantly less coding effort, resulting in a performance and productivity win.

We can run the pMatlab code on any distributed cluster that has the pMatlab and MatlabMPI libraries. Figure 7 shows the basic run script, which requires setting the number of processors on which to run, the names of the processors, and the name of the m-file. The additional commands for launch control clean up old jobs and prepare for the new one.

Current work involves adding Octave into the kernel layer to produce an open source solution as well as expanding the suite of Matlab-compatible message-passing libraries. We’ve also begun work on developing parallel Web services that have pMatlab at their core. For more information on pMatlab and MatlabMPI, visit www.ll.mit.edu/pMatlab and www.ll.mit.edu/MatlabMPI. Note that MatlabMPI is included in the pMatlab suite of software, along with the libraries and example code.

Acknowledgments
This work is sponsored by the US Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the US government.

References

Julie Mullen is a technical staff subcontractor at MIT Lincoln Laboratory. Her research interests include parallel and distributed computing, and high-productivity software tools for engineering applications. Mullen has a PhD in engineering from Brown University. Contact her at jsm@ll.mit.edu.

Nadya Bliss is a technical staff member at MIT Lincoln Laboratory. Her research interests include parallel and distributed computing—specifically, program analysis and optimization, intelligent/cognitive algorithms, and software/hardware co-design methodology. Bliss has an MS in computer science from Cornell University. Contact her at nt@ll.mit.edu.

Robert Bond is the leader of the Embedded Digital Systems group at MIT Lincoln Laboratory. His research interests include research and development of high-performance embedded processors, advanced signal processing, and novel embedded middleware architectures. Bond has a BS in physics from Queen’s University. Contact him at rbond@ll.mit.edu.

Jeremy Kepner is a senior staff member at MIT Lincoln Laboratory. His research interests include parallel and distributed computing—specifically, program analysis and optimization, intelligent/cognitive algorithms, and software/hardware co-design methodology. Kepner has a PhD in astrophysics from Princeton University. Contact him at kepner@ll.mit.edu.

Hahn Kim is an associate staff member at the MIT Lincoln Laboratory. His research interests are in high-performance embedded systems for signal processing and high productivity technologies for parallel computing. Kim has an MS in computer science and engineering from the University of Michigan. Contact him at hkg@ll.mit.edu.

Albert Reuther is a technical staff member at MIT Lincoln Laboratory. His research interests include rapid prototyping and signal processing using high-performance computing and the economics of high-performance computing. Reuther has a PhD in electrical and computer engineering from Purdue University. Contact him at reuther@ll.mit.edu.