

Introduction to Parallel Programming and pMatlab v0.7

Hahn Kim, Julia Mullen
{hgk, jsm}@ll.mit.edu
MIT Lincoln Laboratory, Lexington, MA 02420

Abstract

The computational demands of software continue to outpace the capacities of processor and memory technologies, especially in scientific and engineering programs. One option to improve performance is parallel processing. However, despite decades of research and development, writing parallel programs continues to be difficult. This is especially the case for scientists and engineers who have limited backgrounds in computer science. MATLAB®, due to its ease of use compared to other programming languages like C and Fortran, is one of the most popular languages for implementing numerical computations, thus making it an excellent platform for developing an accessible parallel computing framework.

The MIT Lincoln Laboratory has developed two libraries, pMatlab and MatlabMPI, that not only enables parallel programming with MATLAB in a simple fashion, accessible to non-computer scientists. This document will overview basic concepts in parallel programming and introduce pMatlab.

1. Introduction

Even as processor speeds and memory capacities continue to rise, they never seem to be able to satisfy the increasing demands of software; scientists and engineers routinely push the limits of computing systems to tackle larger, more detailed problems. In fact, they often must scale back the size and accuracy of their simulations so that they may complete in a reasonable amount of time or fit into memory. The amount of processing power directly impacts the scale of the problem being addressed.

One option for improving performance is parallel processing. There have been decades of research and development in hardware and software technologies to allow programs to run on multiple processors, improving the performance of computationally intensive applications. Yet writing accurate, efficient, high performance parallel code is still highly non-trivial, requiring a substantial amount of time and energy to master the discipline of high performance computing, commonly referred to as HPC. However, the primary concern of most scientists and engineers is to conduct research, not to write code.

MATLAB® is the dominant programming language for implementing numerical computations and is widely used for algorithm development, simulation, data reduction, testing and evaluation. MATLAB users achieve high productivity because one line of MATLAB code can typically replace multiple lines of C or Fortran code. In addition, MATLAB supports a range of operating systems and processor architectures, providing portability and flexibility. Thus MATLAB provides an excellent platform on which to create an accessible parallel computing framework.

The MIT Lincoln Laboratory has been developing technology to allow MATLAB users to benefit from the advantages of parallel programming. Specifically, two MATLAB libraries –

This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government

pMatlab and MatlabMPI – have been created to allow MATLAB users to run multiple instances of MATLAB to speed up their programs. This document is targeted towards MATLAB users who are unfamiliar with parallel programming. It provides a high-level introduction to basic parallel programming concepts, the use of pMatlab to parallelize MATLAB programs and a more detailed understanding of the strengths of global array semantics, the programming model used by pMatlab to simplify parallel programming.

The rest of this paper is structured as follows. Section 2 explains the motivation for developing technologies that simplify parallel programming, such as pMatlab. Section 3 provides a brief overview of parallel computer hardware. Section 4 discusses the difficulties in achieving high performance computation in parallel programs. Section 5 introduces the single-program multiple-data model for constructing parallel programs. Section 6 describes the message-passing programming model, the most popular parallel programming model in use today, used by MatlabMPI. Section 7 describes a global array semantics, a new type of parallel programming model used by pMatlab and Section 8 provides an introduction to the pMatlab library and how to write parallel MATLAB programs using pMatlab. Section 9 discusses how to measure performance of parallel programs and Section 10 concludes with a summary.

2. Motivation

As motivation for many of the concepts and issues to be presented in this document, we begin by posing the following question: what makes parallel programming so difficult? While we are familiar with serial programming, the transition to thinking in parallel can easily become overwhelming. Parallel programming introduces new complications not present in serial programming such as:

- Keeping track of which processor data is located on.
- Determining which processor data needs to be delivered to or received from.
- Distributing data such that communication between processors is minimized.
- Distributing computation such that computation on each processor is maximized.
- Synchronizing data between processors (if necessary).
- Debugging and deciphering communication between processors.

Consider transposing a matrix. Let A be a 4×4 matrix. In a serial program, the value $A_{i,j}$ is swapped with the value $A_{j,i}$. In a parallel program, matrix transpose is much more complicated. Let A be distributed column-wise across four processors, such that column n resides on processor $n-1$, e.g. $A_{1,1}$, $A_{2,1}$, $A_{3,1}$ and $A_{4,1}$ are on processor 0, $A_{1,2}$, $A_{2,2}$, $A_{3,2}$ and $A_{4,2}$ are on processor 1, and so on. (Numbering processors starting at 0 is common practice.) To correctly transpose the matrix, data must “move” between processors. For example, processor 0 sends $A_{2,1}$, $A_{3,1}$ and $A_{4,1}$ to processors 1, 2 and 3 and receives $A_{1,2}$, $A_{1,3}$ and $A_{1,4}$ from processors 1, 2 and 3, respectively. Figure 1 depicts a parallel transpose.

Clearly, the increase in speed and memory associated with parallel computing requires a change in the programmer’s way of thinking; she must now keep track of where data reside and where data needs to be moved to. Becoming a proficient parallel programmer can require years of practice. The goal of pMatlab is to hide these details, to provide an environment where MATLAB programmers can benefit from parallel processing without focusing on these details.

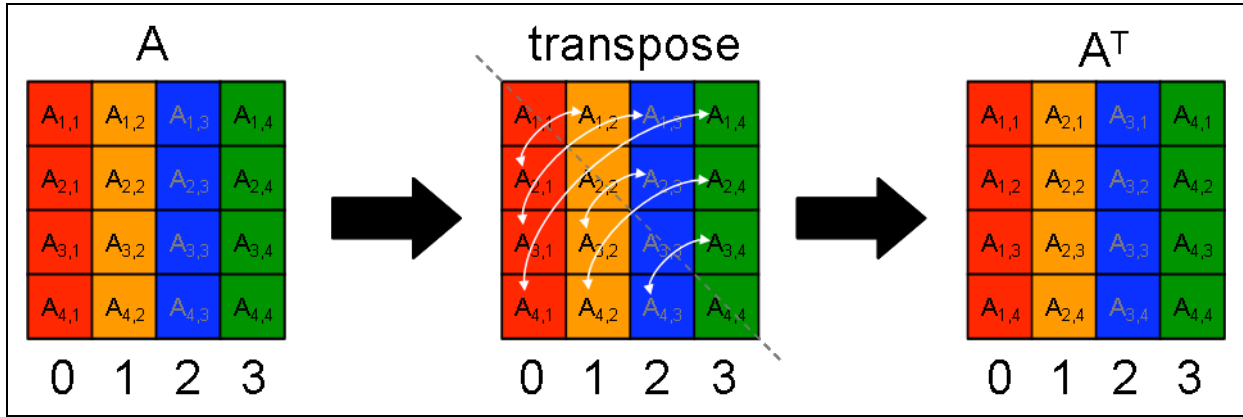


Figure 1 – Example of a parallel transpose. Each column resides on a different processor, represented by a different color. Swapping a column with a row requires communicating with multiple processors.

3. Parallel Computers

Any discussion on parallel programming is incomplete without a discussion of parallel computers. An exhaustive discussion of parallel computer architecture is outside the scope of this paper. Rather, this section will provide a very high-level overview of parallel hardware technology.

There are numerous ways to build a parallel computer. On one end of the spectrum is the *multiprocessor*, in which multiple processors are connected via a high performance communication medium and are packaged into a single computer. Many modern “supercomputers,” such as those made by Cray and IBM, are multiprocessors composed of dozens to hundreds to even thousands of processors linked by high-performance interconnects. Twice a year, the TOP500 project ranks the 500 most powerful computers in the world [1]. The top ranked computer on the TOP500 list as of June 2005 is IBM’s Blue Gene/L at Lawrence Livermore National Laboratory [2]. Blue Gene/L had 65,536 processors, with plans to increase that number even further. Even many modern desktop machines are multiprocessors, supporting two CPUs.

On the other end is the *Beowulf cluster* which uses commodity networks (e.g. Gigabit Ethernet) to connect commodity computers (e.g. Dell) [3]. The advantage of Beowulf clusters is their lower costs and greater availability of components. Performance of Beowulf clusters has begun to rival that of traditional “supercomputers.” In November 2003, the TOP500 project ranked a Beowulf cluster composed of 1100 dual-G5 Apple XServe’s at Virginia Tech the 3rd most powerful computer in the world [4]. MATLAB supports many different operating systems and processor architectures; thus, components for building a Beowulf cluster to run pMatlab and MatlabMPI are relatively cheap and widely available. Figure 2 shows the differences between multiprocessors and Beowulf clusters.

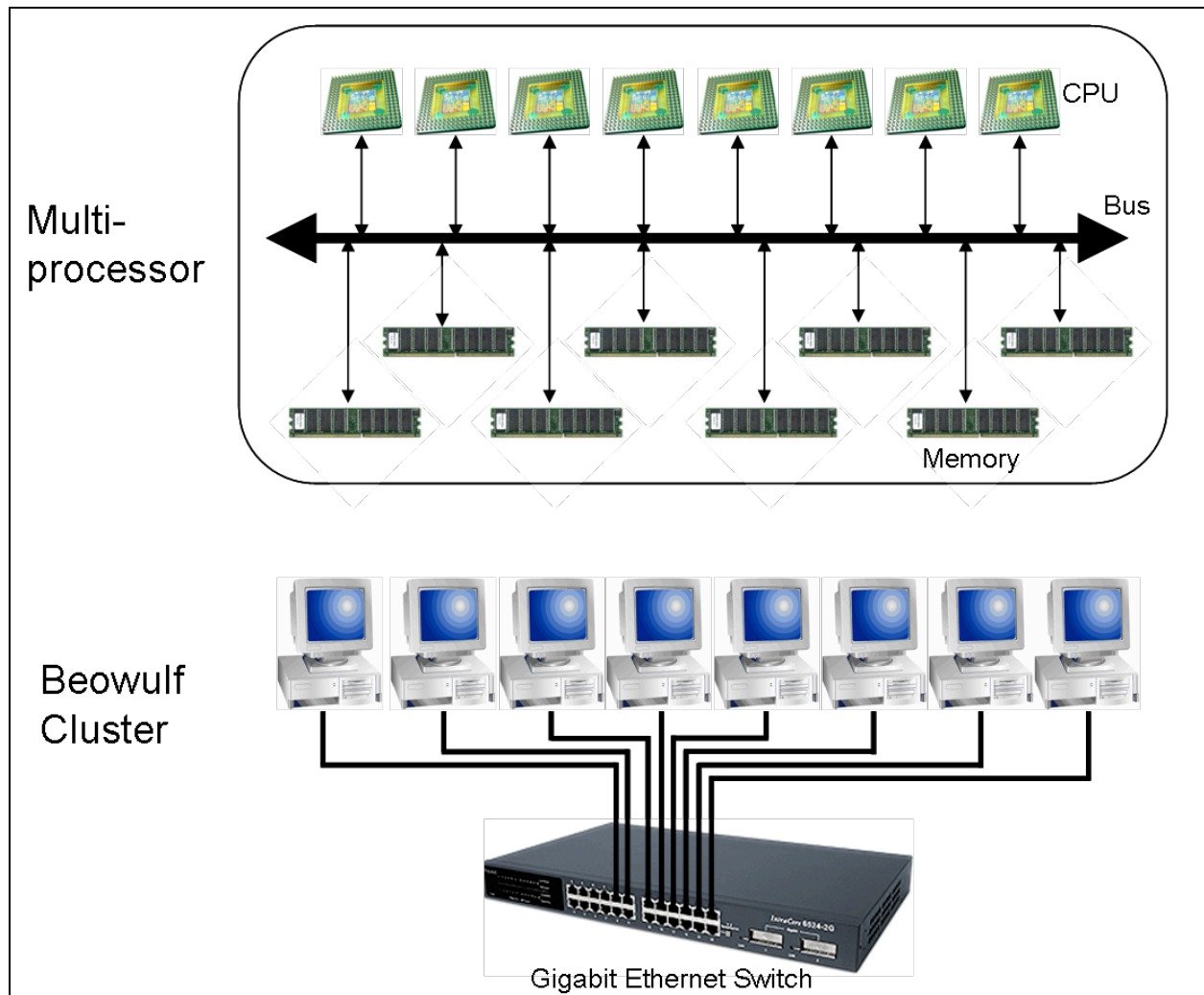


Figure 2 – Multiprocessors often connect processors with high-performance interconnects in a single computer. Beowulf clusters connect commodity computers with conventional networks.

4. Computation and Communication

Parallel programming improves performance by breaking down a problem into smaller sub-problems that are distributed to multiple processors. Thus, the benefits are two-fold. First, the total amount of computation performed by each individual processor is reduced, resulting in faster computation. Second, the size of the problem can be increased by using more memory available on multiple processors.

At first glance, it appears that adding more processors will cause a parallel program to keep running faster, right? Not necessarily. A common misconception by programmers new to parallel programming is that adding more processors always improves performance of parallel programs. There are several basic obstacles that restrict how much parallelization can improve performance.

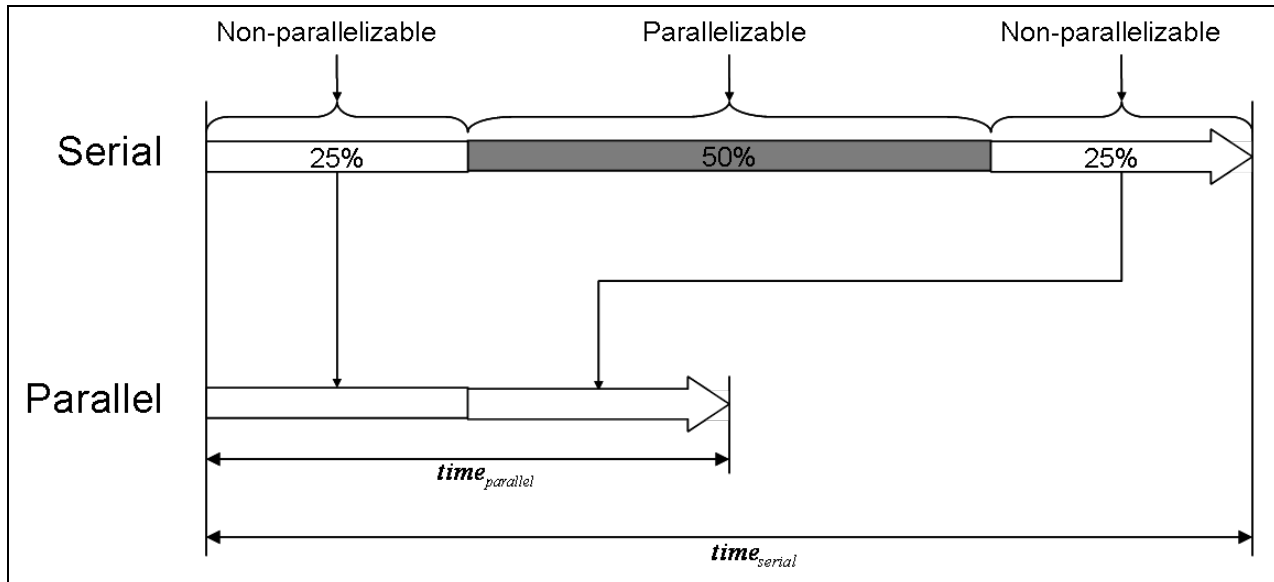


Figure 3 – Serial runtime for an application is shown on top. Fifty percent of the runtime is parallelizable. Amdahl's Law states that even if the parallel section is reduced to 0, total runtime will be reduced by 50%.

4.1. Amdahl's Law

Amdahl's Law was originally defined for processor architectures, but applies equally well to parallel programming. Suppose 50% of a program's runtime can be parallelized. Amdahl's Law states that even if the runtime of the parallelized portion can be reduced to zero, total runtime is only reduced by 50%, as shown in Figure 3. Thus, Amdahl's Law computes the theoretical maximum performance improvement for parallel programs. See Section 9 for more details.

4.2. Division of labor

Dividing and distributing a problem to multiple processors incurs an overhead absent in serial programs. Parallel programmers accept this overhead since, ideally, the time required for this overhead is much less than the time saved due to parallelization. However, as the number of processors increases, so does the overhead. Eventually, this overhead can become larger than the time saved. Programmers must strike a balance between the time spent on overhead and the time saved from parallelizing to maximize the net time saved. Much of the overhead incurred by distributing a problem is caused by communication, which is discussed in the next section.

4.3. Communication

Almost all parallel programs require some communication. Clearly communication is required to distribute sub-problems to processors. However, even after a problem is distributed, communication is often required during processing, e.g. redistributing data or synchronizing processors. Regardless, communication adds overhead to parallel processing. As the number of processors grows, work per processor decreases but communication can – and often does – increase. It is not unusual to see performance increase, peak, then decrease as the number of processors grows, due to an increase in communication. This is referred to as *slowdown* or *speeddown* and can be avoided by carefully choosing the number of processors such that each processor has enough work relative to the amount of communication. Finding this balance is not generally intuitive; it often requires an iterative process.

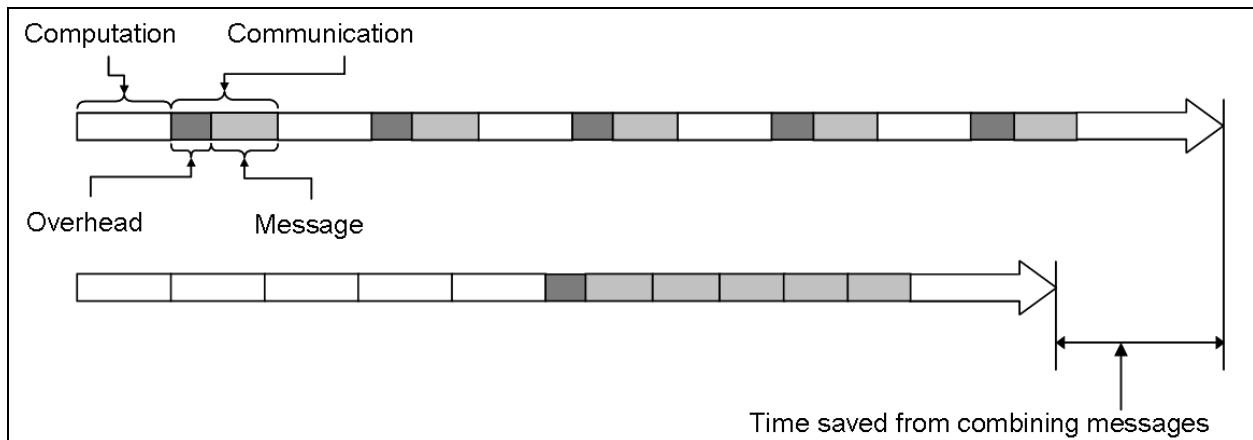


Figure 4 – Example of how combining messages to reduce communication overhead can improve performance incurring communication overhead once rather than multiple times.

One way to mitigate the effects of communication on performance is to reduce the total number of individual messages by sending a few large messages rather than sending many small messages. All networking protocols incur a fixed amount of overhead when sending or receiving a message. Rather than sending multiple messages, it is often worthwhile to wait and send a single large message, thus incurring the network overhead once rather than multiple times as depicted in Figure 4. Depending on the communication medium, overhead may be large or small. In the case of pMatlab and MatlabMPI, overhead per message can be large; therefore, it is to the user’s advantage not only to reduce the total amount of communication but also reduce the total number of messages, when possible.

5. Single-Program Multiple-Data

There are multiple models for constructing parallel programs. At a very high level, there are two basic models: *multiple-program multiple-data* (MPMD) and *single-program multiple-data* (SPMD).

In the MPMD model, each processor executes a different program (“multiple-program”), with the different programs working in tandem to process different data (“multiple-data”). Streaming signal processing chains are excellent examples of applications that can be parallelized using the MPMD model. In a typical streaming signal processing chain, data streams in and flows through a sequence of operations. In a serial program, each operation must be processed in sequential order on a single processor. In a parallel program, each operation can be a separate program executing simultaneously on different processors. This allows the parallel program to process multiple sets of data simultaneously, in different stages of processing.

Consider an example of a simulator that generates simulated data to test a signal processing application, shown in Figure 5. The simulator is structured as a *pipeline*, in which data flow in one direction through various stages of processing. Stages in the pipeline can be executed on different processors or a single stage could even execute on multiple processors. If two consecutive stages each run on multiple processors, it may be necessary to redistribute data from one stage to the next; this is known as a *cornerturn*.

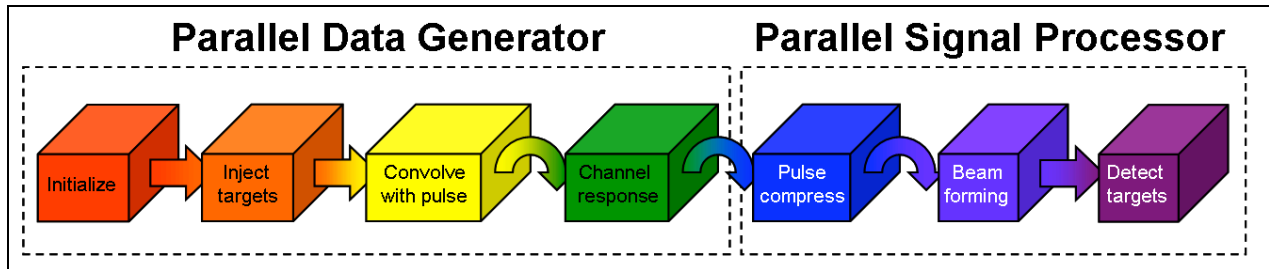


Figure 5 – Example of a MPMD signal processing chain. Each box and color represents a stage in the pipeline and unique set of processors, respectively. Curved arrows indicate cornerturns.

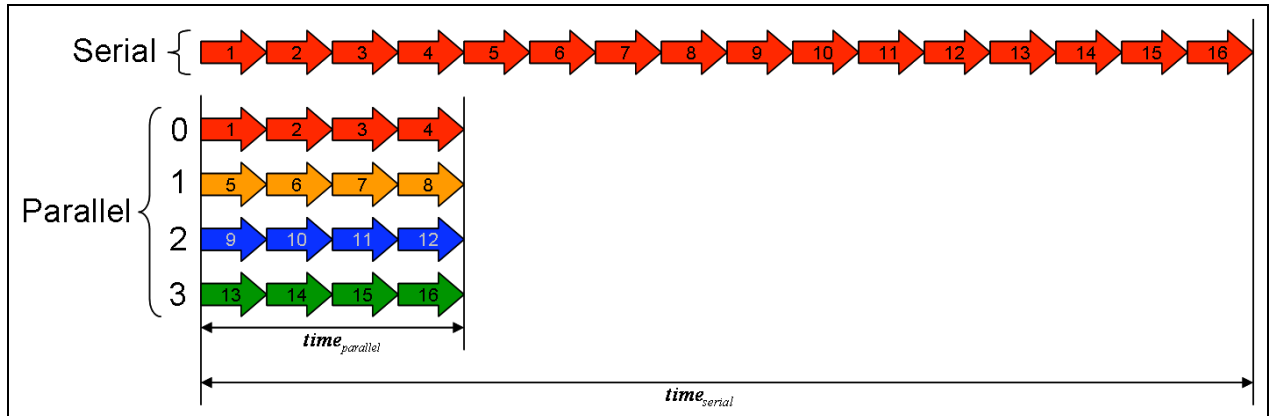


Figure 6 – Each arrow represents one Monte Carlo simulation. The top set of arrows represents serial execution of the simulations. The bottom set of arrows represent parallel execution of the simulations.

Note that a pipeline does not necessarily speed up processing a single frame. However, since each stage executes on different processors, the pipeline can process multiple frames simultaneously, increasing throughput.

In the SPMD model, the same program runs on each processor (“single-program”) but each program instance processes different data (“multiple-data”). Different instances of the program may follow different paths through the program (e.g. if-else statements) due to differences in input data, but the same basic program executes on all processors. pMatlab and MatlabMPI follow the SPMD model. Note that SPMD programs can be written in a MPMD manner by using if-else statements to execute different sections of code on different processors. In fact, the example shown in Figure 5 is actually a pMatlab program written in an MPMD manner. The scalability and flexibility of the SPMD model has made it the dominant parallel programming model. SPMD codes range from tightly coupled (*fine-grain parallelism*) simulations of evolution equations predicting chemical processes, weather and combustion, to “embarrassingly parallel” (*coarse-grain*) programs searching for genes, data strings, and patterns in data sequences or processing medical images or signals.

Monte Carlo simulations are well suited to SPMD models. Each instance of a Monte Carlo simulation has a unique set of inputs. Since the simulations are independent, i.e. the input of one is not dependent on the output of another, the set of simulations can be distributed among processors and execute simultaneously, as depicted in Figure 6.

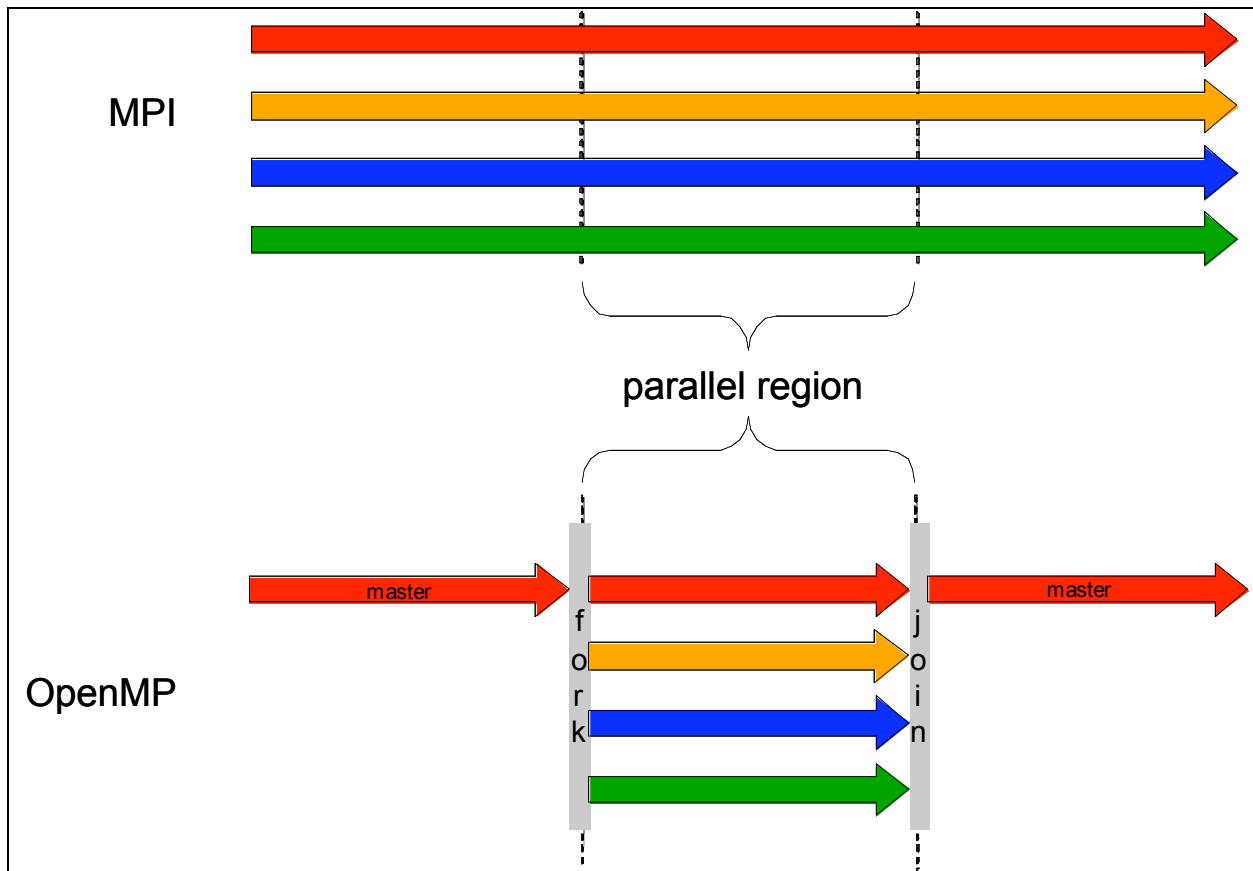


Figure 7 – In OpenMP, threads are spawned at parallel regions of code marked by the programmer. In MPI, every process runs the entire application.

The two most common implementations of the SPMD model are the Message Passing Interface (MPI) and OpenMP. In MPI, the program is started on all processors. The program is responsible for distributing data among processors, then each processor processes its section of the data, communicating among themselves as needed. MPI is an industry standard and is implemented on a wide range of parallel computers, from multiprocessor to cluster architectures [5]. Consequently, MPI is the most popular technology for writing parallel SPMD programs. More details of MPI are discussed in the next section.

The OpenMP model provides a more dynamic approach to parallelism by using a *fork-join* approach [6]. In OpenMP, programs start as a single process known as a *master thread* that executes on a single processor. The programmer designates *parallel regions* in the program. When the master thread reaches a parallel region, a *fork* operation is executed that creates a *team* of threads, which execute the parallel region on multiple processors. At the end of the parallel region, a *join* operation terminates the team of threads, leaving only the master thread to continue on a single processor.

pMatlab and MatlabMPI use the MPI model, thus the code runs on all processors for the entire runtime of the program. Figure 7 graphically depicts the difference between pMatlab/MatlabMPI and OpenMP programs. At first, this may seem unnecessarily redundant. Why bother executing

code in regions that do not require parallelization on multiple processors that will generate the same result? Why not execute only the parallelized code on multiple processors, like OpenMP?

Consider initialization code (which is typically non-parallel) that initializes variables to the same value on each processor. Why not execute the code on a single processor, then send the values to the other processors? This incurs communication which can significantly reduce performance. If each processor executes the initialization code, the need for communication is removed. Using a single processor for non-parallel code doesn't improve performance or reduce runtime of the overall parallel program.

The fork-join model works with OpenMP because OpenMP runs on shared-memory systems, in which every processor uses the same physical memory. Therefore, communication between threads does not involve high latency network transactions but rather low-latency memory transactions.

6. Message Passing

Within the SPMD model, there are multiple styles of programming. One of the most common styles is *message passing*. In message passing, all processes are independent entities, i.e. each process has its own private memory space. Processes communicate and share data by passing

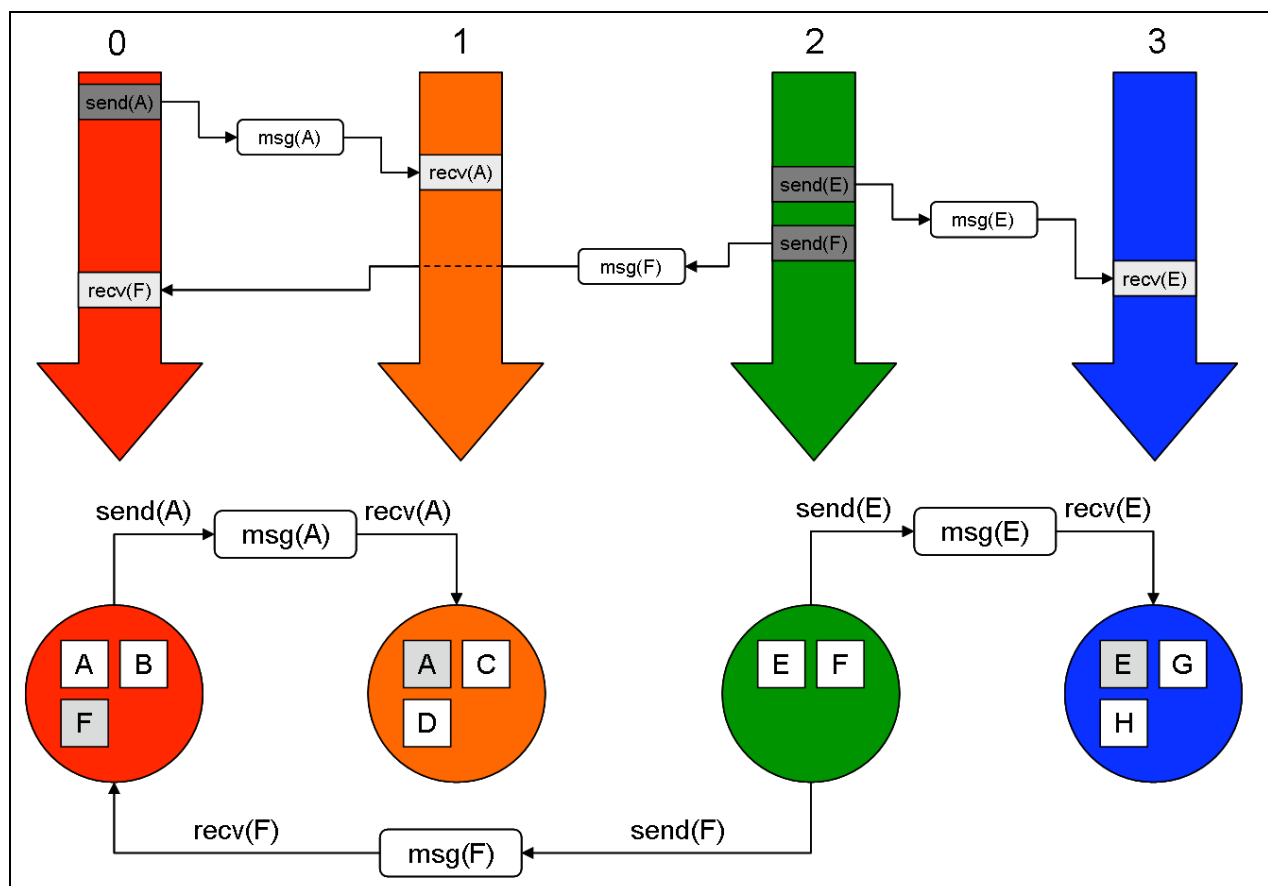


Figure 8 – Example of message passing. Each color represents a different processor. The upper half represents program flow on each processor. The lower half depicts each processor's memory space.

messages. Communication is performed explicitly by directly invoking send and receive operations in the code, as shown in Figure 8. The most popular message passing technology is the Message Passing Interface (MPI), a message passing library for C and Fortran. MPI was developed with the goal of creating a message passing standard that would be portable across a wide range of systems, from multiprocessors to Beowulf clusters.

MPI is a basic networking library tailored to parallel systems, enabling processes in a parallel program to communicate with each other. Details of the underlying network protocols and infrastructure are hidden from the programmer. This helps achieve MPI's portability mandate while enabling programmers to focus on writing parallel code rather than networking code.

To distinguish sub-processes from each other, each process is given a unique identifier called a *rank*. The rank is the primary mechanism MPI uses to distinguish processors from each other. Ranks are used for various purposes. MPI programs use ranks to identify which data should be processed by each process or what operations should be run on each process. The following are a couple examples:

- Executing different sections of code

```
if (rank == 0)
    % Rank 0 executes the if block
else
    % All other ranks execute the else block
end
```

- Processing different data

```
% Use rank as an index into 'input' and 'output' arrays
output(rank) = my_function(input(rank));
```

6.1. MatlabMPI

MatlabMPI is a MATLAB implementation of the MPI standard developed at the Lincoln Laboratory to emulate the “look and feel” of MPI [7]. MatlabMPI is written in pure MATLAB code (i.e. no MEX files) and uses MATLAB's file I/O functions to perform communication, allowing it to run on any system that MATLAB supports. MatlabMPI has been used successfully both inside and outside Lincoln to obtain speedups in MATLAB programs previously not possible. Figure 9 depicts a message being sent and received in MatlabMPI.

Despite MPI's popularity, writing parallel programs with MPI and MatlabMPI can still be difficult. Explicit communication requires careful coordination of sending and receiving messages and data that must be divided among processors can require complex index computations. The next section describes how message passing can be used to create a new parallel programming model, called global array semantics, that hides the complexity of message passing from the programmer.

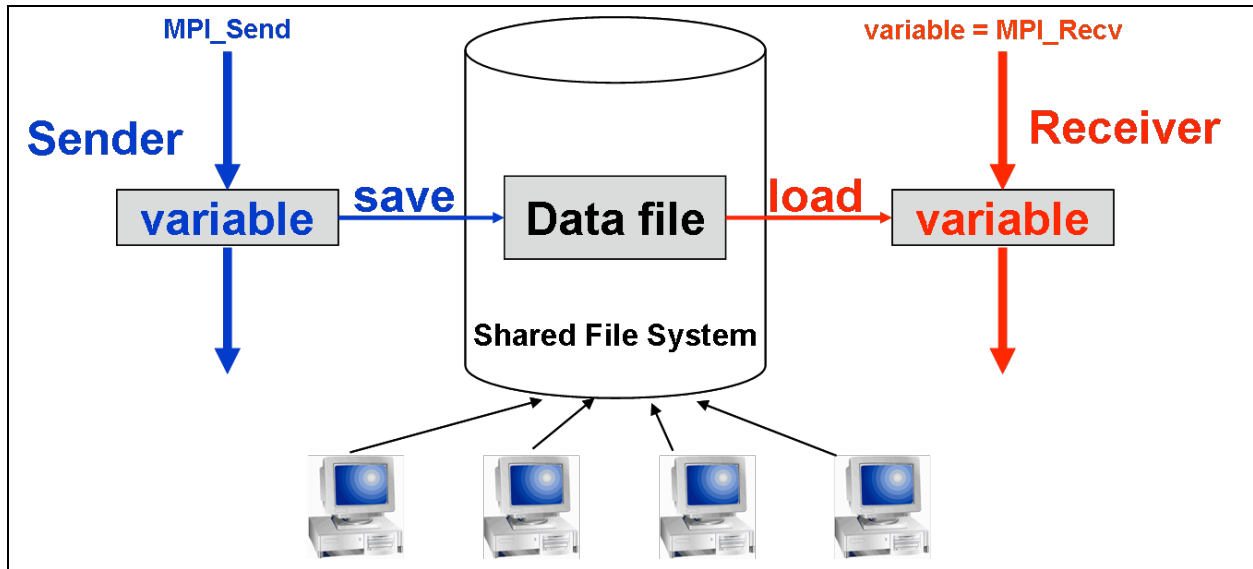


Figure 9 – MatlabMPI implements the fundamental communication operations in MPI using MATLAB's file I/O functions.

7. Global Array Semantics

Message passing provides an extremely flexible programming paradigm that allows programmers to parallelize a wide range of data structures, including arrays, graphs, and trees. However, data in signal processing applications are generally stored in regular data structures, e.g. vectors, matrices, tensors, etc.

Global array semantics is a parallel programming model in which the programmer views an array as a single global array rather than multiple, independent arrays located on different processors, e.g. MPI. The ability to view related data distributed across processors as a single array more closely matches the serial programming model, thus making the transition from serial to parallel programming much smoother.

A global array library can be implemented using message passing libraries such as MPI or MatlabMPI. The program calls functions from the global array library. The global array library determines if and how data must be redistributed and calls functions from the message passing library to perform the communication. Communication is hidden from the programmer; arrays are automatically redistributed when necessary, without the knowledge of the programmer. Just as message passing libraries like MPI allow programmers to focus on writing parallel code rather than networking code, global array libraries like pMatlab allow programmers to focus on writing scientific and engineering code rather than parallel code.

7.1. pMatlab

pMatlab brings global array semantics to MATLAB, using the message passing capabilities of MatlabMPI [8]. The ultimate goal of pMatlab is to move beyond basic messaging (and its inherent programming complexity) towards higher level parallel data structures and functions, allowing any MATLAB user to parallelize their existing program by simply changing and adding a few lines, rather than rewriting their program, as would likely be the case with MatlabMPI.

See Figure 10 for a comparison of implementing an FFT and cornerturn in MatlabMPI and pMatlab.

Clearly, pMatlab allows the scientist or engineer to access the power of parallel programming while focusing on the science or engineering. Figure 11 depicts the hierarchy between pMatlab applications, the pMatlab and MatlabMPI libraries, MATLAB and the parallel hardware.

```

if (my_rank==0) | (my_rank==1) | (my_rank==2) | (my_rank==3)
    A_local=rand(M,N/4);
end

if (my_rank==4) | (my_rank==5) | (my_rank==6) | (my_rank==7)
    B_local=zeros(M/4,N);
end

tag = 0;
if (my_rank==0) | (my_rank==1) | (my_rank==2) | (my_rank==3)
    A_local=fft(A_local);
    for ii = 0:3
        MPI_Send(ii+4, tag, comm, A_local(ii*M/4 + 1:(ii+1)*M/4,:));
    end
end

if (my_rank==4) | (my_rank==5) | (my_rank==6) | (my_rank==7)
    for ii = 0:3
        B_local(:, ii*N/4 + 1:(ii+1)*N/4) = MPI_Recv(ii, tag, comm);
    end
end

mapA = map([1 4], {}, [0:3]);
mapB = map([4 1], {}, [4:7]);
A = rand(M,N,mapA);
B = zeros(M,N,mapB);
B(:, :) = fft(A);

```

Figure 10 – MatlabMPI (top) and pMatlab (bottom) implementations of an FFT and cornerturn.

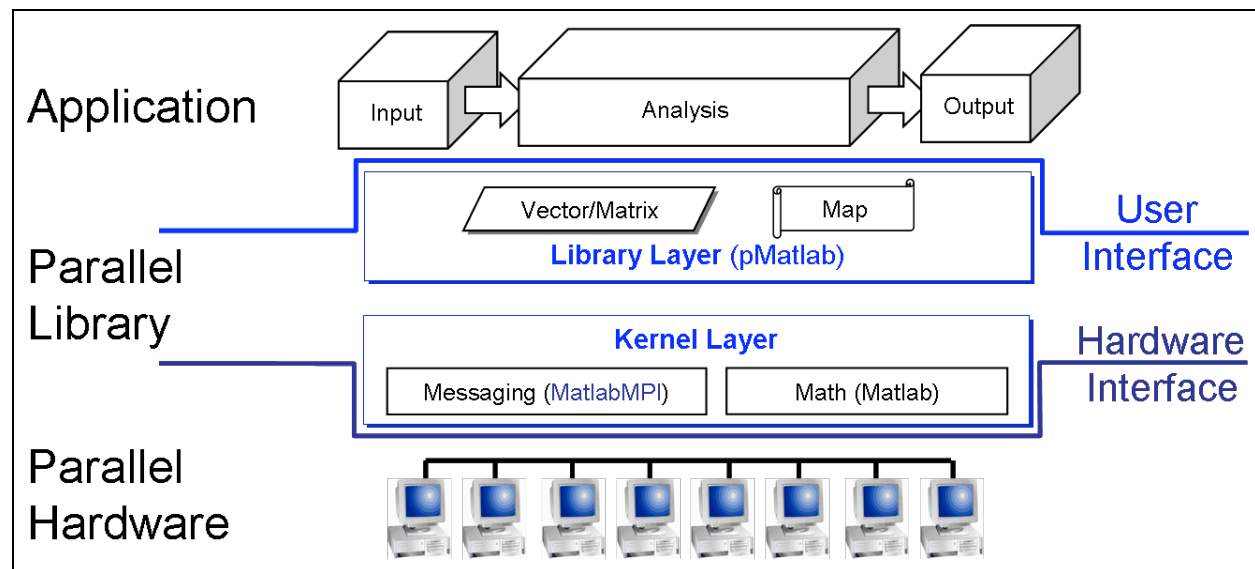


Figure 11 – Parallel MATLAB consists of two layers. pMatlab provides parallel data structures and library functions. MatlabMPI provides messaging capability.

```

D = ones(M, P);           % Create a MxM dmat object of ones with map P
D = zeros(M, N, R, P);    % Create a MxNxR dmat object of zeros using
                           % map P
D = rand(M, N, R, S, P);  % Create a MxNxRxS dmat object of random
                           % numbers with map P
D = spalloc(M, N, X, P);  % Create a MxN sparse dmat with room to hold
                           % X non-zero values with map P

```

Figure 12 – Examples of `dmat` constructor functions.

8. Introduction to pMatlab

This section introduces the fundamentals of pMatlab: creating data structures; writing and launching applications; and testing, debugging and scaling your application. A tutorial for pMatlab can be found in [9]. For details on functions presented here and examples, refer to [10].

8.1. Distributed Matrices

pMatlab introduces a new datatype: the *distributed matrix*, or *dmat*. The `dmat` is the fundamental data storage datatype in pMatlab, equivalent to `double` in MATLAB. pMatlab can support `dmat` objects with two, three, or four dimensions. `dmat` objects must be explicitly constructed in pMatlab via a *constructor function*.

Currently, pMatlab overloads four MATLAB constructor functions: `zeros`, `ones`, `rand`, and `spalloc`. Each of these functions accepts the same set of parameters as their corresponding MATLAB functions, with the addition of a `map` parameter. The `map` parameter accepts a `map` object which describes how to distribute the `dmat` object across multiple processors.

Figure 12 contains examples of how to create `dmats` of varying dimensions. Section 8.2 describes `map` objects and how to create them in greater detail. Section 8.3 discusses how to program with `dmats`. Refer to [10] for details on each constructor function.

8.2. Maps

For a `dmat` object to be useful, the programmer must tell pMatlab *how* and *where* the `dmat` must be distributed. This is accomplished with a *map* object. To create a `dmat`, the programmer first creates a `map` object, then calls one of several constructor functions, passing in the `dmat`'s dimensions and the `map` object.

Each `map` object has four components:

- The *grid* is a vector of integers that specifies how each dimension of a `dmat` is broken up. For example, if the `grid` is `[2 3]`, the first dimension is broken up between 2 processors and the second dimension is broken up between 3 processors, as shown in Figure 13.
- The *distribution* specifies how to distribute each dimension of the `dmat` among processors. There are three types of distributions:
 - *Block* – Each processor contains a single contiguous block of data
 - *Cyclic* – Data are interleaved among processors
 - *Block-cyclic* – Contiguous blocks of data are interleaved among processors. Note that block and cyclic are special cases of block-cyclic. Block-cyclic distributions require the user to also specify the block size.

Figure 14 shows an example of the same `dmat` distributed over four processors using each of the three types of data distributions.

- The *processor list* specifies on which ranks the object should be distributed. Ranks are assigned column-wise (top-down, then left-right) to grid locations in sequential order (Figure 15).
- *Overlap* is a vector of integers that specify the amount of data overlap between processors for each dimension. Only block distributions can have overlap. Overlap is useful in situations when data on the boundary between two processors are required by both, e.g. convolution. In these cases, using overlap copies boundary data onto both processors.

Figure 16 shows an example of a `dmat` distributed column-wise across four processors with 1 column of overlap between adjacent processors.

The following are examples of how to create various types of maps:

- 2D map, 1×2 grid, block along rows, cyclic along columns:

```
grid1 = [1 2];           % 1x2 grid
dist1(1).dist = 'b';      % block along dim 1 (rows)
dist1(2).dist = 'c';      % cyclic along dim 2 (columns)
proc1 = [0:1];           % list of ranks 0 and 1
map1 = map(grid1, dist1, proc1);
```

- 3D map, 2×3×2 grid, block-cyclic along rows and columns with block size 2, cyclic along third dimension:

```
grid2 = [2 3 2];         % 2x3x2 grid
dist2(1).dist = 'bc';     % block-cyclic along dim 1 (rows)
dist2(1).b_size = 2;      % block size 2 along dim 1 (rows)
dist2(2).dist = 'bc';     % block-cyclic along dim 2 (columns)
dist2(2).b_size = 2;      % block size 2 along dim 2 (columns)
dist2(3).dist = 'c';      % cyclic along dim 3
proc2 = [0:3];           % list of ranks 0 through 3
map2 = map(grid2, dist2, proc2);
```

- 2D map, 2×2 grid, block along both dimensions, overlap in the column dimension of size 1 (1 column overlap). This example shows how to specify a map in a single line of code:

```
% param 1: 2x2 grid
% param 2: block in all dimensions
% param 3: list of ranks 0 and 1
% param 4: overlap of 0 along dim 1, overlap of 1 along dim 2
map3 = map([2 2], {}, [0 1], [0 1]);
```

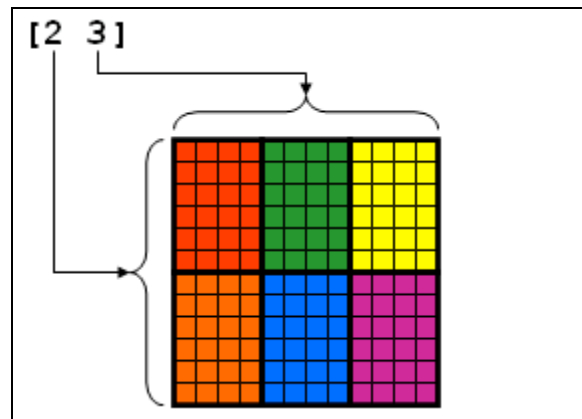


Figure 13 – Example of a map grid.

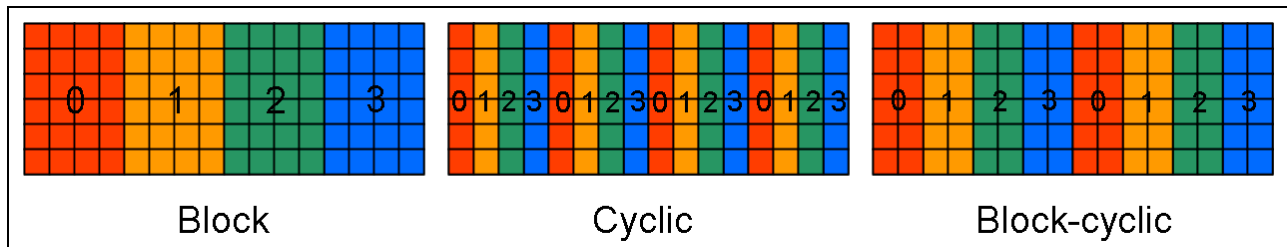


Figure 14 – Examples of different data distributions.

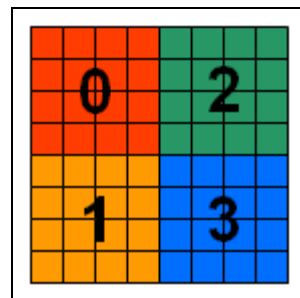


Figure 15 – Example of how processors are assigned to grid locations.

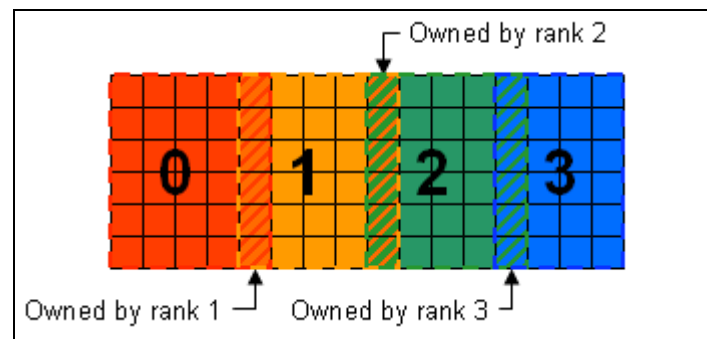


Figure 16 – Example of data overlap between processors.

8.3. Working with Distributed Matrices

Using MATLAB's function overloading feature, pMatlab is able to provide new implementations of existing MATLAB functions. This capability was demonstrated in Section 8.1. However, pMatlab does not limit itself to overloading just constructors, but other functions as well. These overloaded functions provide the same functionality as their MATLAB equivalents but operate on the `dmats` rather than `doubles`. In this way, pMatlab provides an interface that is nearly identical to MATLAB while abstracting the parallel computation and communication from the user. While some functions require additional input, such as the `map` parameter for the `dmat` constructor functions discussed in Section 8.1, most functions look identical to their MATLAB counterparts.

pMatlab overloads a number of basic MATLAB functions. Most of these overloaded functions implement only a subset of the available functionality of each function. For example, MATLAB's `zeros` function allow matrix dimensions to be specified by arbitrarily long vectors while the pMatlab `zeros` function does not.

Occasionally, pMatlab users may wish to directly manipulate a `dmat` object. Thus, pMatlab provides additional functions that allow the user to access the contents of `dmat` object without using overloaded pMatlab functions.

This section will introduce various classes of functions available in pMatlab and touch on the major differences between MATLAB and pMatlab functions.

8.3.1. Subscripted Assignment and Reference

Two of the most fundamental MATLAB operations are subscripted assignment (`subsasgn`) and reference (`subsref`). Initially, both operators appear similar in that they are both invoked by the parentheses operator, `()`, and access subsections of a matrix. However, the degrees to which they duplicate MATLAB's built-in operations are substantially different.

```
N = 1000000;

% A and B have different maps
mapA = map([1 Ncpus], {}, 0:Ncpus-1);
mapB = map([Ncpus 1], {}, 0:Ncpus-1);
A = zeros(N, mapA);
B = rand(N, mapB);

% Invokes pMatlab's subsasgn.
% Data in B is remapped to mapA and assigned to A.
A(:, :) = B;

% Performs a memory copy.
% A is replaced with a copy of B, i.e. A has the same data AND mapping
% as B.
A = B
```

Figure 17 – Examples of when pMatlab's `subsasgn` is and is not invoked.


```

% Fully supported
A(:, :) % Entire matrix reference
A(i, j) % Single element reference

% Partially supported
A(i:k, j:l) % Arbitrary submatrix reference

```

Figure 18 – Examples of fully supported and partially supported cases **subsref** in pMatlab.

Subscripted assignment

pMatlab's `subsasgn` operator supports nearly all capabilities of MATLAB's built-in operator. pMatlab's `subsasgn` can assign any arbitrary subsection of a `dmat`. If a subsection spans multiple processors, `subsasgn` will automatically write to the indices on the appropriate processors.

`subsasgn` in pMatlab has one restriction. In MATLAB, the parentheses operator can be omitted when assigning to the entire matrix. In pMatlab, the parentheses operator **must** be used in this case. Performing an assignment without `()` simply performs a memory copy and the destination `dmat` will be identical – including the mapping – to the source; data will not be redistributed. See Figure 17 for an example.

Subscripted reference

The capabilities of pMatlab's `subsref` operator are much more limited than `subsasgn`. `subsref` returns a “standalone” `dmat` only in the following cases:

- `A(:, :)` – Refers to the entire contents of the `dmat` `A`
- `A(i, j)` – Refers to a single element in the `dmat` `A`.

Referring to an arbitrary subsection will not produce a “standalone” `dmat`, i.e. the resulting `dmat` can not be directly used as an input to any pMatlab function, with the exception of `local`. The details are outside the scope of this document; this is a limitation of pMatlab and addressing these issues requires further research. For more information about the `local` function, see Section 8.3.4. Figure 18 contains examples of fully and partially supported cases of `subsref`.

8.3.2. Arithmetic Operators

A number of arithmetic operators, such as addition (+), matrix multiplication (*), element-wise multiplication (. *), and equality (==) have been overloaded. These operators are used the same way as MATLAB. Figure 19 show several examples of various arithmetic operators in pMatlab.

8.3.3. Mathematical Functions

Several mathematical functions, including simple functions such as absolute value (`abs`) and advanced functions such as `fft` have been overloaded in pMatlab. Most of these functions are called the same way as in MATLAB. Some pMatlab functions may not support the entire syntax supported by the equivalent MATLAB function, but all will support at least the basic syntax. Figure 20 shows several examples of how to use various mathematical functions in pMatlab.

```

N = 1000000;
M1 = map([1 Ncpus], {}, 0:Ncpus-1);
M2 = map([Ncpus 1], {}, 0:Ncpus-1);

% Add two dmats
A = rand(N, M1);      % NxN dmat mapped to M1
B = rand(N, M1);      % NxN dmat mapped to M1
C1 = A + B;           % Result is mapped to M1

C2 = zeros(N, M2);    % NxN dmat mapped to M2
C2(:, :) = A + B;     % Result is remapped to M2

% Matrix multiply dmat and a double
D = rand(N, M1);      % NxN dmat mapped to M1
E = rand(N);          % NxN double
F1 = D * E;           % Result is mapped to M1

F2 = zeros(N, M2);    % NxN dmat mapped to M2
F2(:, :) = D * E;     % Result is remapped to M2

% Check for equality between two dmats
G = rand(N, M1);      % NxN dmat mapped to M1
H = rand(N, M1);      % NxN dmat
I1 = (G == H);        % Result is mapped to M1

I2 = zeros(N, M2);    % NxN dmat mapped to M2
I2(:, :) = (G == H); % Result is remapped to M2

```

Figure 19 – Examples of how to use arithmetic operators with dmats

```

N = 1000000;
M1 = map([1 Ncpus], {}, 0:Ncpus-1);
M2 = map([Ncpus 1], {}, 0:Ncpus-1);

% Absolute value
A = rand(N, M1);      % NxN dmat mapped to M1
B1 = abs(A);          % Result is mapped to M1

B2 = zeros(N, M2);    % NxN dmat mapped to M2
B2(:, :) = abs(A);    % Result is remapped to M2

% FFT
D = rand(N, M1);      % NxN dmat mapped to M1
E1 = fft(D);          % Result is mapped to M1

E2 = zeros(N, M2);    % NxN dmat mapped to M2
E2(:, :) = fft(D);    % Result is remapped to M2

```

Figure 20 – Examples of how to use mathematical functions with dmats

8.3.4. Global vs. Local Scope

The notion of matrices distributed across multiple processors raises the issue of global and local scopes of a `dmat`. The global scope of a `dmat` refers to the matrix as a single entity, while local scope refers to only the section of the `dmat` residing on the local processor. `pMatlab` includes various functions that specifically address the global and local scopes of `dmats`. These functions allow the user a greater degree of control over `dmat` objects, but must be used with caution.

The `local` and `put_local` functions give the user direct access to the local contents of `dmat` objects. Each processor in a `dmat`'s map contains a section of the `dmat`. `local` returns a matrix that contains the section of the `dmat` that resides on the local processor. Conversely, `put_local` writes the contents of a matrix into that processor's section of the `dmat`. The `local` and `put_local` functions allow the user to perform operations not implemented in `pMatlab` while still taking advantage of `pMatlab`'s ability to distribute data and computation.

When `local` returns the local portion of a `dmat`, the resulting matrix loses global index information, i.e. which indices in the global map it maps to. See Figure 21 for an example. Sometimes it is necessary for each processor to know which global indices it owns. `pMatlab` contains functions that compute what global indices each processor owns for a given `dmat`:

- `global_block_range/global_block_ranges` - Returns the first and last indices of a continuous block of indices owned by local processor and all processors, respectively. Works for block distributed `dmats` only.
- `global_ind/global_inds` - Returns the indices owned by the local processor and all processors, respectively.
- `global_range/global_ranges` - Returns the first and last indices for multiple blocks of indices owned by the local processor and all processors, respectively. Works for all distributions.

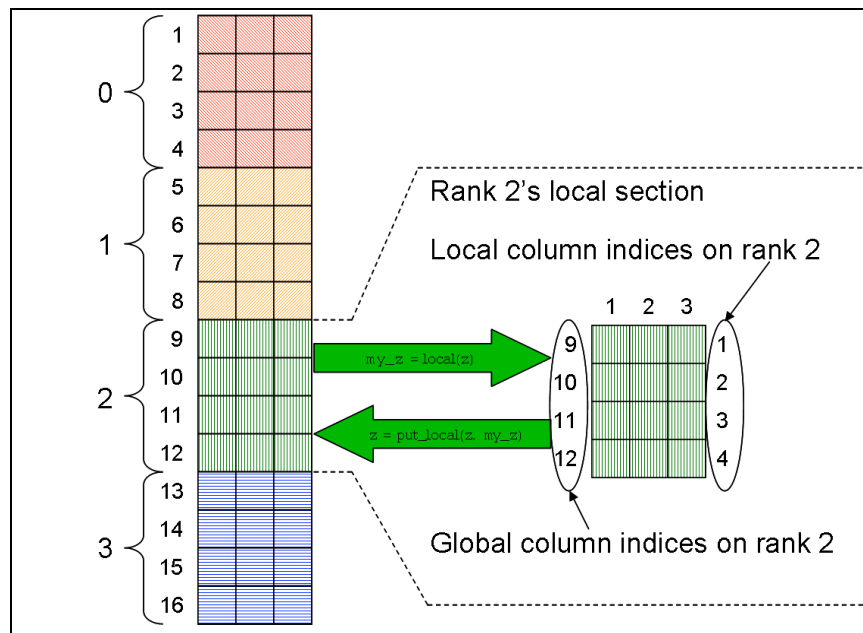


Figure 21 – Comparison of global and local indices in a `dmat`.

```

1:  % RUN.m is a generic script for running pMatlab scripts.
2:
3:  % Define number of processors to use
4:  Ncpus = 4;
5:
6:  % Name of the script you want to run
7:  mFile = 'sample_application';
8:
9:  % Define cpus
10:
11: % Run on user's local machine
12: % cpus = {};
13:
14: % Specify which machines to run on
15: % cpus = {'node1.ll.mit.edu', ...
16:          'node2.ll.mit.edu', ...
17:          'node3.ll.mit.edu', ...
18:          'node4.ll.mit.edu'};
19:
20: % Abort left over jobs
21: MPI_Abort;
22: pause(2.0);
23:
24: % Delete left over MPI directory
25: MatMPI_Delete_all;
26: pause(2.0);
27:
28: % Define global variables
29: global pMATLAB;
30:
31: % Run the script.
32: ['Running: ' mFile ' on ' num2str(Ncpus) ' cpus']
33: eval(MPI_Run(mFile, Ncpus, cpus));

```

Figure 22 – Example script for launching pMatlab applications.

8.4. Launching pMatlab

The recommended method of launching pMatlab programs is to use a launch script similar to the one in Figure 22. The launch script **must** be located and run in the same directory as the pMatlab application.

Line 4 defines the `Ncpus` variable, which specifies the how many processors to run on.

Lines 7 defines the `mfile` variable, which specifies which pMatlab program to run. `mfile` should be set to the filename of the pMatlab program *without* the `.m` extension.

Lines 12 and 15-18 are examples of different ways to define the `cpus` variable, which specifies where to launch the pMatlab program. The user should uncomment the line(s) he wishes to use.

- Line 12 directs pMatlab to launch all MATLAB processes on the user's local machine; this is useful for debugging purposes, but `Ncpus` should be set to a small number (i.e. less than or equal to 4) to prevent overloading the local machine's processor.
- Lines 15-18 directs pMatlab to launch the MATLAB processes on the machines in the list. If `Ncpus` is smaller than the list, pMatlab will use the first `Ncpus` machines in the list. If `Ncpus` is larger than the list, pMatlab will wrap around to the beginning of the list.

These are the basic ways to launch pMatlab; for a more details, please refer to `MPI_Run` in [10].

Line 21 looks in the current working directory for a `MatMPI` directory. Each time a pMatlab application is launched, a `MatMPI` directory is created that contains various scripts and files used to launch the application. If `MatMPI` exists, a pMatlab program has been launched from this directory in the past and may still be running. Since multiple instances of pMatlab can not run simultaneously from the same directory, `MPI_Abort` is run to kill any currently running pMatlab program. Line 22 pauses to allow the system to reach a quiescent state after `MPI_Abort`.

Line 25 removes any existing `MatMPI` directory. Line 26 pauses to allow the filesystem to reach a quiescent state after `MatMPI_Delete_all`.

Line 29 declares the `pMATLAB` variable global. `pMATLAB` is a data structure used by pMatlab.

Line 32 displays a statement indicating the pMatlab program is about to be launched.

Line 33 launches the pMatlab program specified by `mfile` on `Ncpus` processors, using the machines specified in `cpus`.

Note that program specific parameters can not be passed into pMatlab applications, via `MPI_Run`. Program parameters should be implemented as variables initialized at the beginning of the pMatlab application specified in `mfile`, such as line 1 in the pMatlab code shown in Figure 23, which will be explained in the next section.

8.5. A Simple pMatlab Application

This section will describe a very simple pMatlab application that performs a parallel FFT shown in Figure 23. This example can be found in the `examples` directory in the pMatlab library.

Line 1 sets the size of the input matrix to the FFT.

Line 4 enables or disables the pMatlab library. If `PARALLEL` is set to 0, then the script will not call any pMatlab functions or create any pMatlab data structures and will run serially on the local processor. If `PARALLEL` is set to 1, then the pMatlab library will be initialized, `dmats` will be created instead of regular MATLAB matrices, and any functions that accept `dmat` inputs will call the overloaded pMatlab functions.

Line 7 initializes the maps, `mapX` and `mapY`, for the input and output matrices, respectively. If `PARALLEL` is set to 0, then `mapX` and `mapY` are set to 1.

```

1:  N = 2^10; % NxN Matrix size.
2:
3:  % Turn parallelism on or off.
4:  PARALLEL = 1; % Can be 1 or 0. OK to change.
5:
6:  % Create Maps.
7:  mapX = 1; mapY = 1;
8:  if (PARALLEL)
9:      % Initialize pMatlab.
10:     pMatlab_Init;
11:     Ncpus = pMATLAB.comm_size;
12:     my_rank = pMATLAB.my_rank;
13:
14:     % Break up channels.
15:     mapX = map([1 Ncpus], {}, 0:Ncpus-1);
16:     mapY = map([1 Ncpus], {}, 0:Ncpus-1);
17: end
18:
19: % Allocate data structures.
20: X = rand(N,N,mapX);
21: Y = zeros(N,N,mapY);
22:
23: % Do fft. Changes Y from real to complex.
24: Y(:,,:) = fft(X);
25:
26: % Finalize the pMATLAB program
27: disp('SUCCESS');
28: if (PARALLEL)
29:     pMatlab_Finalize;
30: end

```

Figure 23 – A parallel FFT implemented in pMatlab.

If `PARALLEL` is set 1, then lines 8 through 17 are executed. Line 10 initializes the pMatlab library. Line 11 obtains the number of processors the program is executing. Line 12 obtains the local processor's rank. Lines 15 and 16 create map objects for the `dmats` `x` and `y`. The maps distribute the `dmats` column-wise.

Lines 20 and 21 construct the `dmats` `x` and `y`. `x` is an `N×N dmat` of random values and `y` is an `N×N dmat` of zeros.

Line 24 calls `fft` on `x` and assigns the output to `y`. Since `x` is a `dmat`, rather than call the built-in `fft`, MATLAB calls pMatlab's `fft` that is overloaded for `dmat`.

Lines 28 through 30 finalizes the execution of the pMatlab library if `PARALLEL` is set to 1.

8.6. Interactive Use of pMatlab

Before discussing pMatlab's interactive capabilities, it is important that we define “interactive” with respect to MATLAB and pMatlab and how these definitions differ. In MATLAB, the user

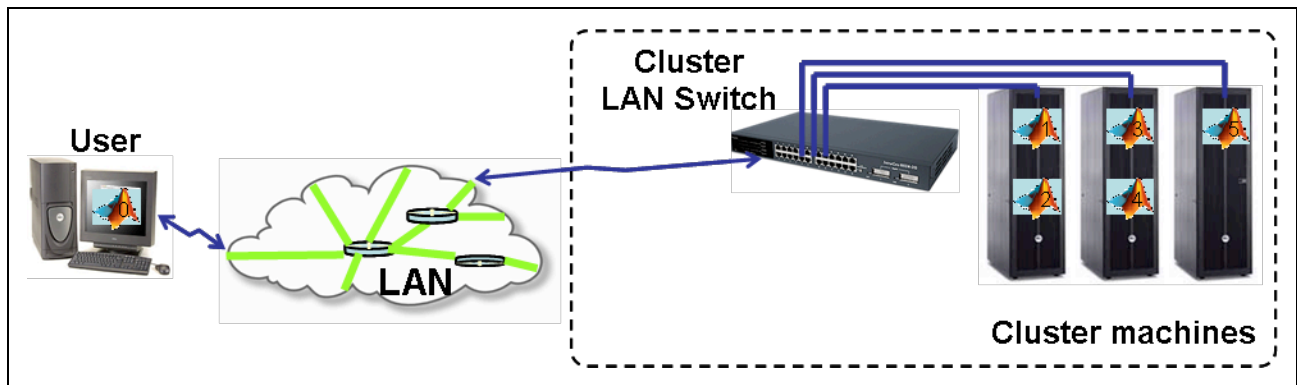


Figure 24 – Interactively running pMatlab applications on both the user’s computer and a cluster of machines. Rank 0 runs on the user’s machine and the remaining ranks run on the cluster.

command prompt or by invoking scripts and functions that contain MATLAB code. The former is MATLAB’s definition of “interactive.”

In pMatlab, the user can issue pMatlab commands only by invoking scripts or functions that contain pMatlab code. **pMatlab commands can not be run directly from the command prompt.** Thus, pMatlab does not support MATLAB’s notion of “interactive” use.

However, pMatlab does support a different type of “interactivity.” Traditionally, running parallel programs requires logging into a “submit” node of a parallel computer, writing a submit script (containing the program name, parameters, and number of processors), and submitting the script to a *job scheduler*. Once submitted, jobs often must wait in a *job queue* before they are executed. The job scheduler e-mails the user when the program has finished. This process is known as *batch processing* and is, by nature, highly non-interactive.

While pMatlab still supports limited batch processing, it introduces a level of interactivity to parallel programming previously unseen. To launch pMatlab jobs, a user first starts MATLAB on his personal machine, then runs the `RUN.m` script, such as the one presented in Figure 22. When the pMatlab job starts it not only launches MATLAB processes on remote machines but it also uses the existing MATLAB process, known as the *leader process*, on user’s personal machine as well, as shown in Figure 24. Consequently, unlike the traditional batch process model of submitting a parallel job, waiting for a notification of completion, transferring files to a desktop and then post-processing the results, pMatlab users are able to use their personal display and utilize MATLAB’s graphics capabilities as the job processes or completes.

8.7. Developing, Testing, Debugging, and Scaling

Though pMatlab removes a great deal of the complexity associated with parallel code development, the process is still inherently more difficult than serial coding. As with serial coding there are software development strategies which lead to simplified code development. In addition, parallel coding raises the issues of scaling to more processors and receiving debugging information from multiple nodes, which do not arise in serial implementations.

8.7.1. Developing

There are several guidelines that pMatlab users should follow when developing pMatlab applications. We list the most important below.

Write code that can be run independent of pMatlab

A general tenet of good software engineering is that computer code should be as modular as possible. The easier it is to enable and disable portions of the simulation, the easier it is to pinpoint the section which holds an error. In addition, the ability to quickly remove and replace a specific module allows for easy updates of existing code.

pMatlab code is no exception and should be written in such a manner that the pMatlab library can be easily disabled to run the application serially, i.e. on a single processor. Developing applications in this manner assists the user in locating bugs. In parallel computing there are generally two types of errors, those resulting from algorithmic choice and those resulting from parallel implementation. In all cases it is recommended that the code be run in serial first to resolve any algorithmic issues. After the code has been debugged in serial, any new bugs in the parallel code can be assumed to arise from the parallel implementation. Being able to easily enable and disable the pMatlab library can help the user determine if a bug is caused by the algorithm or by parallelization. Figure 22 provides an example of this development approach, using the `PARALLEL` variable.

Write scalable code

Avoid writing code that depends on a specific number of processors. When writing parallel code, remember that along with speedup you gain the ability to work on larger problems. The problem that fits on four processors today may grow to fit on eight next year and so on. If your pMatlab application requires a fixed number of processors, it is impossible to scale the simulation up (or down) without editing, rebuilding and retesting the code. In most cases writing scalable pMatlab code is relatively straightforward as the parallelism is implicitly controlled by the library rather than explicitly controlled by the user, unlike libraries such as MPI and MatlabMPI.

The above practice of avoiding the use of hard coded numbers of processors applies to all aspects of the code. The more the code is parameterized the less rewriting, rebuilding, retesting is required. Thus, when initializing pMatlab, make sure to obtain the number of processors from the `pMATLAB` variable and store it to a variable like `Ncpus`. Use this variable when creating maps in order to create scalable, flexible code. See Figure 22 for an example.

Avoid rank-dependent code

Traditionally parallel programmers had to keep track of the data distribution, and occasionally needed to create rank dependent code, e.g. writing if-else statements that execute different sections of code on different processors. However, pMatlab has been designed to alleviate all of that for the programmer. The main philosophy behind pMatlab is to hide the parallelism from the user as much as possible. The motivation for pMatlab was to relieve programmers of the burden of explicitly parallelizing their code as required in MPI and MatlabMPI.


```

% Let D be of type dmat.
% D_agg contains entire contents of D on rank 0.
% D_agg is empty on all other ranks.
D_agg = agg(D);

if (my_rank == 0)
    % Process D_agg on rank 0
end

% Save "results" variable to "results.<rank>.mat"
filename = ['results.' num2str(my_rank)];
save(filename, 'results');

```

Figure 25 – Examples of “safe” rank-dependent pMatlab code.

pMatlab users should not need to write code that depends on the rank and should indeed avoid it as rank-dependent code can potentially break or cause unexpected behavior in pMatlab programs.

There are some exceptions to this rule. One is related to the `agg` function which returns the entire contents of a `dmat` to the leader processor but returns only the local portion of matrix to the remaining processors. There may be cases where the user wishes to operate on the aggregated matrix without redistributing it. In this case, the code that operates on the aggregated matrix might be within an `if` block than runs on only the leader processor. Note that rank 0 is chosen as the leader processor because there will always be a rank 0, even when running on a single processor.

The other main exception to writing rank dependent code is associated with I/O. If the user wishes each process in the pMatlab program to save its results to a `.mat` file, then each processor must save its results to a unique filename. The simplest way to accomplish this is to include the rank in the filename as each rank is unique. Figure 25 provides examples of “safe” rank-dependent pMatlab code.

8.7.2. Testing

When a serial MATLAB program fails, the user is notified directly via an error message (and usually a beep) at the MATLAB command prompt. Failures in pMatlab programs can be much more devious and subtle. A failure in just one MATLAB process can cause the entire program to hang. For instance, suppose a pMatlab program contains a `cornerturn`, which requires all processes to communicate with all other processors. If one MATLAB process fails prior to the `cornerturn`, the remaining processes will hang while waiting for a message that will never arrive from the failed process. All non-leader MATLAB processes redirect their output – and error messages – to `.out` files in the `MatMPI` directory. If the error occurs on a non-leader process, the user will receive no notification of the error. This is a common occurrence in many pMatlab programs. **Users should make a habit of routinely checking the contents of the `MatMPI/* .out` files when they suspect their programs have hung.**

Improving performance via parallelism is no excuse for inefficient code. Parallel programs can also benefit from profiling to identify bottlenecks in both serial and parallel sections.

8.7.3. Debugging

The MATLAB debugger can be used to assist in debugging pMatlab programs in a limited fashion. The debugger runs on only the leader process on the user's machine. Nevertheless, even this limited debugging capability can provide valuable information to the user.

Since pMatlab is SPMD, every processor should execute every line of code. However, since the debugger only runs on the leader process, breakpoints do not affect non-leader processes. Non-leader processes will simply continue past breakpoints set by the user until they reach a parallelized operation which requires communication with the leader. However, this asynchronous behavior will not affect the behavior of the program or the debugger.

8.7.4. Scaling

Another tenet of good software engineering is that programs should not be run on full scale inputs immediately. Rather, programs should initially be run on a small test problem to verify functionality and to validate against known results. The programs should be scaled to larger and more complicated problems until the program is fully validated and ready to be run at full scale. The same is true for parallel programming. Parallel programs should not run at full scale on 32 processors as soon as the programmer has finished taking a first stab at writing the application. Both the test input and number of processors should be gradually scaled up.

The following is the recommended procedure for scaling up a pMatlab application. This procedure gradually adds complexity to running the application.

1. **Run with 1 processor on the user's local machine with the pMatlab library disabled.** This tests the basic serial functionality of the code.
2. **Run with 1 processor on the local machine with pMatlab enabled.** Tests that the pMatlab library has not broken the basic functionality of the code.
3. **Run with 2 processors on the local machine.** Tests the program's functionality works with more than one processor without network communication.
4. **Run with 2 processors on multiple machines.** Test that the program works with network communication.
5. **Run with 4 processors on multiple machines.**
6. **Increase the number of processors, as desired.**

Figure 26 shows the sequence of parameters that should be used to scaling pMatlab applications. See Figure 22 and Figure 23 for examples on how to set these parameters.

	In pMatlab code	In RUN.m	
1.	PARALLEL = 0;	Ncpus = 1;	cpus = {};
2.	PARALLEL = 1;	Ncpus = 1;	cpus = {};
3.	PARALLEL = 1;	Ncpus = 2;	cpus = {};
4.	PARALLEL = 1;	Ncpus = 2;	cpus = {'node1', 'node2'};
5.	PARALLEL = 1;	Ncpus = 4;	cpus = {'node1', 'node2'};
6.	PARALLEL = 1;	...	cpus = {'node1', 'node2'};

Figure 26 – Example sequence of parameters for scaling parallel programs.

9. Measuring Performance

This section covers some of the basic methods to measure and compute performance of parallel programs.

9.1. Speedup

The most common measure of performance for parallel programs is *speedup*. For input size n and number of processors p , speedup is the ratio between the serial and parallel runtimes:

$$speedup(n, p) = \frac{time_{serial}(n)}{time_{parallel}(n, p)}$$

Most speedup curves can be placed into the following categories:

- *Linear* – Occurs when the application runs p times faster on p processors than on a single processor. In general, linear (also known as *ideal*) is the theoretical limit.
- *Superlinear* – In rare cases, a parallel program can achieve speedup greater than linear. This can occur if the subproblems fit into cache or main memory, significantly reducing data access times.
- *Sublinear* – Occurs when adding more processors improves performance, but not as fast as linear. This is a common speedup observed in many parallel programs.
- *Saturation* – Occurs when communication time grows faster than computation time decreases. Often there is a “sweet spot” when performance peaks. This is also a common speedup observed in parallel programs.

If no communication is required, then speedup should be linear with N . These types of applications are known as *embarrassingly parallel*. If communication is required, then the non-communicating portion of the code should speedup linearly with N . Figure 27 shows examples of each of the different types of speedups.

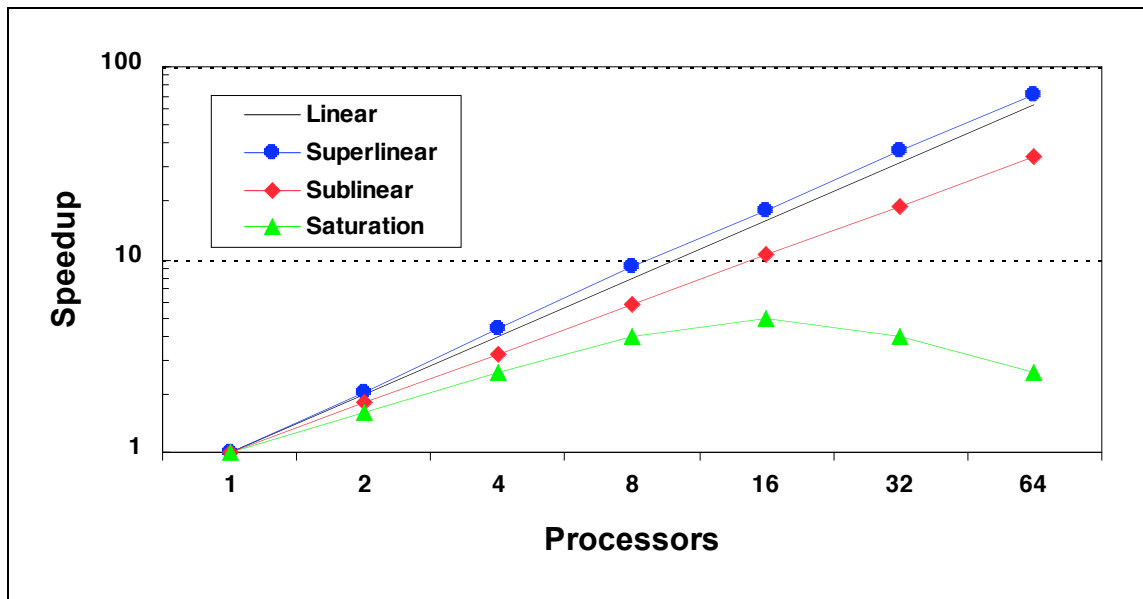


Figure 27 – Examples of typical speedup curves.

9.2. Amdahl's Law

Amdahl's Law is a useful principle for determining the theoretical maximum speedup a parallel program can achieve. As stated in [11], "Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used." In our case, the "faster mode of execution" is parallelizing one or more sections of code. Amdahl's Law can be concisely described using the following equation, where f is the fraction of time spent on serial operations and p is the number of processors:

$$speedup \leq \frac{1}{f + (1 - f)/p}$$

From this equation, we see that as the number of processors increases, the term $(1 - f)/p$ approaches 0, resulting in the following:

$$\lim_{p \rightarrow \infty} speedup \leq \frac{1}{f}$$

As we can see, the maximum possible speedup for a parallel program depends on how much of a program can be parallelized. Consider three programs with three different values of f : 0.1, 0.01, and 0.001. The maximum possible speedups for these programs are 10, 100, and 100, respectively. Figure 28 shows the speedups for these programs up to 1024 processors.

Amdahl's Law clearly demonstrates the law of diminishing returns: as the number of processors increases, the amount of speedup attained by adding more processors decreases.

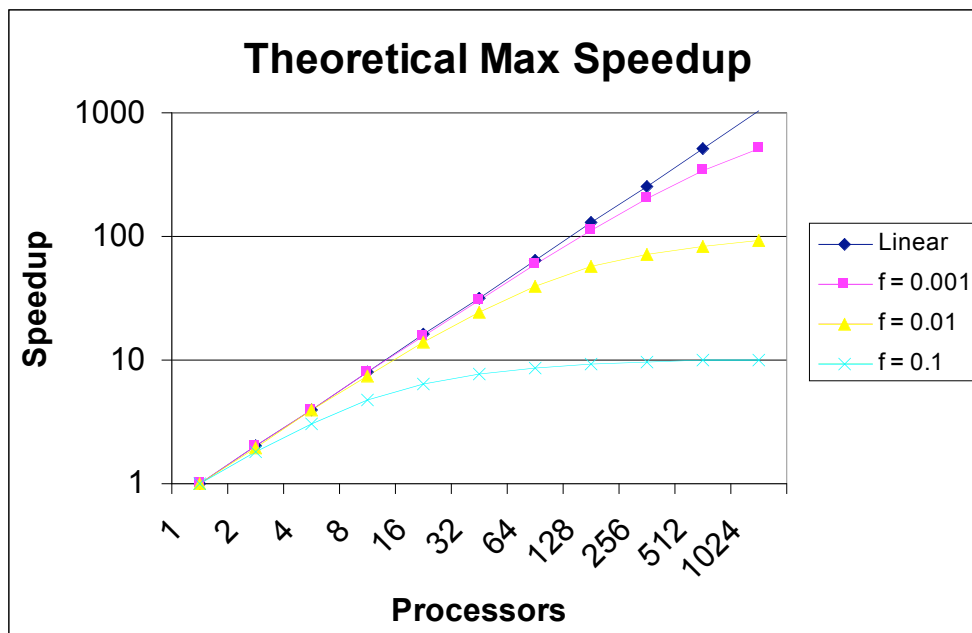


Figure 28 – Theoretical maximum speedups for three parallel programs with different values of f .

One common misconception about speedup is the amount of benefit adding processors provides. With respect to a single processor, 16 processors provide a maximum of 16× speedup and 32 processors provides a maximum of 32× speedup. Obvious, right? What many parallel programmers miss is that while 16 processors provide a speedup of up to 16× over a single processor, adding 16 more processors provides an *additional speedup of only 2×*. Keep this in mind when determining how many resources to use when running a pMatlab program.

10. Conclusion

In this document, we have introduced the following:

- Basic concepts in parallel programming
- Basic programming constructs in pMatlab
- How to launch, write, test, debug, and scale pMatlab applications
- How to measure performance of parallel programs

And yet, this is just the beginning. Before you tackle writing your own pMatlab application, take a moment to look other available pMatlab documentation. For example, many programs fall into a class of applications known as parameter sweep applications in which a set of code, e.g. a Monte Carlo simulation, is run multiple times using different input parameters. [12] introduces a pMatlab template that can be used to easily parallelize existing parameter sweep applications. Additionally, the pMatlab distribution contains an examples directory which contains example programs that can be run and modified.

11. References

- [1] TOP500 webpage. <http://www.top500.org>.
- [2] IBM Blue Gene webpage. <http://www.research.ibm.com/bluegene/>
- [3] T. L. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese. How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters. MIT Press, Cambridge, MA, 1999.
- [4] Virginia Tech’s System X webpage. <http://www.tcf.vt.edu/systemX.html>
- [5] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra. MPI – The Complete Reference: Volume 1, The MPI Core. MIT Press. Cambridge, MA. 1998.
- [6] OpenMP webpage. <http://www.openmp.org>
- [7] J. Kepner. “Parallel Programing with MatlabMPI.” 5th High Performance Embedded Computing workshop (HPEC 2001), September 25-27, 2001, MIT Lincoln Laboratory, Lexington, MA
- [8] J. Kepner and N. Travinin. “Parallel Matlab: The Next Generation.” 7th High Performance Embedded Computing workshop (HPEC 2003), September 23-25, 2003, MIT Lincoln Laboratory, Lexington, MA.
- [9] J. Kepner, A. Reuther, H. Kim. “Parallel Programming in Matlab Tutorial.” MIT Lincoln Laboratory.
- [10] H. Kim, N. Travinin. “pMatlab v0.7 Function Reference.” MIT Lincoln Laboratory.
- [11] D. Patterson, J. Hennessy. “Computer Architecture: A Quantitative Approach.” Morgan Kaufman, 2003.
- [12] H. Kim, A. Reuther. “Writing Parameter Sweep Applications using pMatlab.” MIT Lincoln Laboratory.