L13 – Supercomputing - Part 1

Without question the recent wide spread availability of large scale distributed computing (supercomputing) is revolutionizing the types of problems we are able to solve in all branches of the physical sciences. Currently almost every major university now hosts some kind of supercomputing architecture, and hence most researchers currently have the ability to develop software for such an environment. This is in stark contrast to the situation a decade ago where one had to obtain computing time from dedicated supercomputing centers which were few and far between. This availability of resources is only going to increase in the future and as a result it is important to know the basics of how to develop code and how to utilize supercomputer facilities.

We could actually dedicate an entire seminar series to supercomputing, but in this class we only have two lectures. So, what we will do here is (1) Introduce the primary concepts behind supercomputing, and (2) Introduce the fundamentals of how to actually write code that supercomputers can run. There are many details on the coding aspects that are better suited to a full scale course.

1. What is Supercomputing?

So, what is a supercomputer? Here's a picture of one – the common type of picture you will see on a website. Looks impressive right, a whole room full of ominous looking black boxes just packed with cpu's.

Typical picture of a now obsolete supercomputer.

Here's the official definition of a supercomputer:

- A computer that leads the world in terms of processing capacity, speed of calculation, at the time of its introduction.

My preferred definition is:

- Any computer that is **only one** generation behind what you really need.

So, the definition of a supercomputer is really defined by processing speed. What does this mean for our current supercomputers?

In Terminator 3 Skynet is said to be operating at "60 teraflops per second" either this makes no sense or the speed of Skynets calculations are accelerating.

Computer Speed

Computer speed is measured in **FL**oating Point **O**perations **P**er **S**econd (**FLOPS**). Floating point is way to represent real numbers (not integers) in a computer. As we discussed previously this involves an approximation as we don't have infinite memory locations for our real numbers. We usually represent real numbers by a number of significant digits which we scale using an exponent:

$$\textbf{significant digits} \times \textbf{base}^{\textbf{exponent}}$$

We are generally most familiar with the base 10 system so as an example we could represent the number 1.5 as:

$$1.5 \times 10^0 \text{ , or}$$
$$0.15 \times 10^1 \text{, or}$$
$$0.015 \times 10^2 \text{, etc.}$$

We say floating point because the decimal point is allowed to **float** relative to the significant digits of the number. So, a floating point operation is simply any mathematical operation (addition, subtraction, multiplication, etc.) between floating point numbers.

Currently the **LINPACK Benchmark** is officially used to determine a computers speed. You can download the code and directions yourself from:

http://www.netlib.org/benchmark/hpl

The benchmark solves a dense system of linear equations (**Ax =b**) where the matrix **A** is of size **N × N**. It utilizes a solution based on Gaussian elimination (which every student here should at least recall what that is) that utilizes a numerical approach called partial pivoting. The calculation requires $\frac{2}{3}N^3 + 2N^2$ FLOPS. The benchmark is run for different size matrices (different **N** values) searching for the size **Nmax** where the maximal performance is obtained.

To see the current computing leaders you can check out the website:

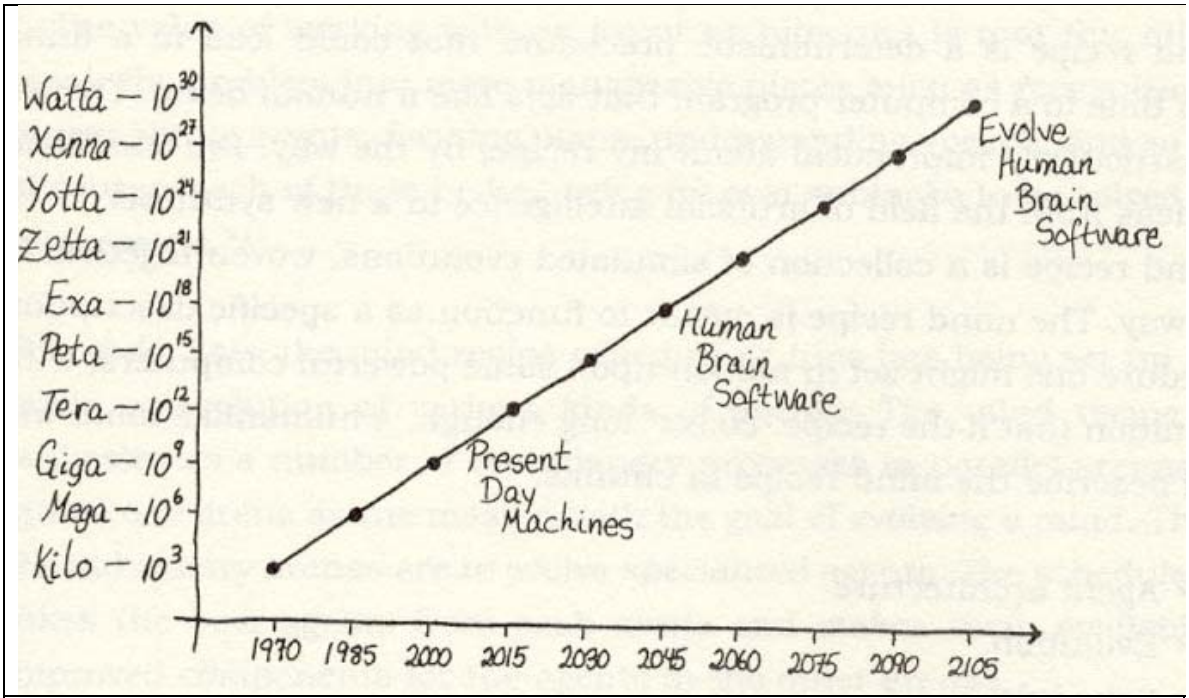
http://www.top500.org

It's truly amazing to look at this. The last time I gave a talk on supercomputing the most recent update to the **Top500** list was posted on Nov. 2006. At this time the computer **BlueGene/L** at Lawrence Livermore National Laboratory (LLNL) was the unchallenged leader with a max performance of 280.6 Tera FLOPS. It's amazing to see how dramatically this has changed. The current leader (June 2010) is the **Jaguar** supercomputer at Oak Ridge National Laboratory which maxes out at **1759 Tera FLOPS**. Blue Gene/L is now at about 480 Tera FLOPS but has dropped to the number 8 position.

The first parallel computers were built in the early 1970's (e.g., Cray's ILIAC IV). But, we can see a pretty linear progression in computing speed:

Year	Speed		Computer
1974	**100**	**Mega FLOPS**	CDC STAR 100 (LLNL)
1984	**2.4**	**Giga FLOPS**	M-13 (Scientific Research Institute, Moscow)
1994	**170**	**Giga FLOPS**	Fujitsu Numerical Wind Tunnel (Tokyo)
2004	**42.7**	**Tera FLOPS**	SGI Project Columbia (NASA)
2006	**280.6**	**Tera FLOPS**	Blue Gene/L (LLNL)
2010	**1759**	**Tera FLOPS**	Jaguar (Oak Ridge National Laboratory)

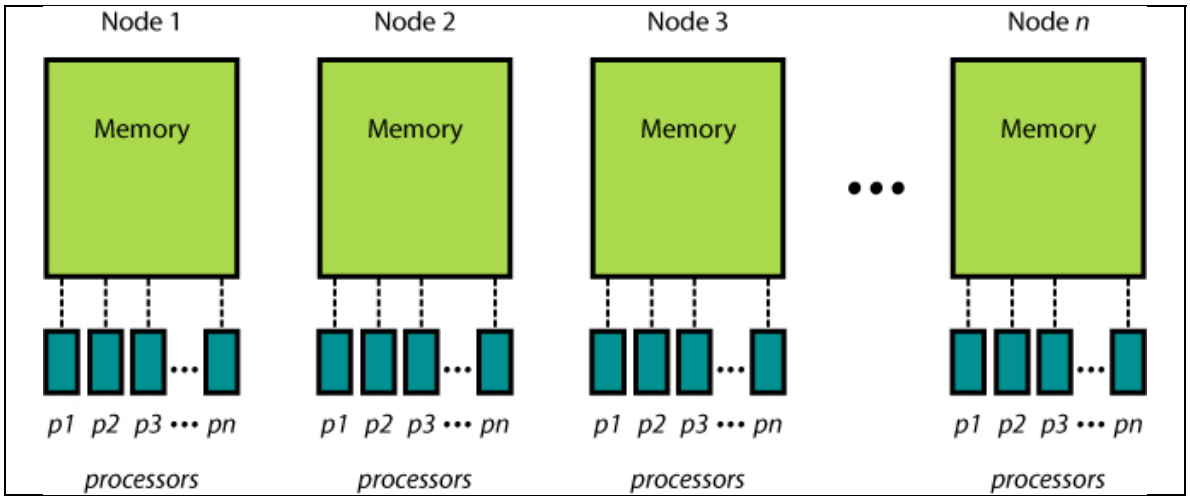
This result is a basic outcome of **Moore's Law** which states that the number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years. The next figure is an interesting look at what may happen if this trend continues.



From Rudy Rucker's book, *The Lifebox, the Seashell, and the Soul: What Gnarly Computation Taught Me About Ultimate Reality, the Meaning of Life, and How to Be Happy.*

2. Parallelism in Physics

To understand why the current model of supercomputing has been so successful we must first look at what this model is. Basically the preferred supercomputer architecture today is called Parallel Computing, which means that we divide our problem up among a number of processors. The following diagram shows the basic computer lay out:



The main points are:

- The computer is divided up into **nodes**.
- Each **node** may have **multiple processors** (E.g., most Linux clusters may have 2 processors per node; but the majority of the computers I've worked on have 8 processors per node).
- Each processor has access to a global memory structure on it's node – but doesn't have access to the memory on the other nodes.
- **Communication** of information can occur between processors within or across nodes.
- Each processor can access all of the memory for each node.

The reason this strategy is so important is because:

The fundamental laws of physics are parallel in nature.

That is, the fundamental laws of physics apply at each point (or small volume) in space. In general we are able to describe the dynamic behavior of physical phenomena by a system(s) of differential equations. Examples are:

- Heat flow
- The Wave Equation
- Mantle Convection
- Hydrodynamics

- etc.

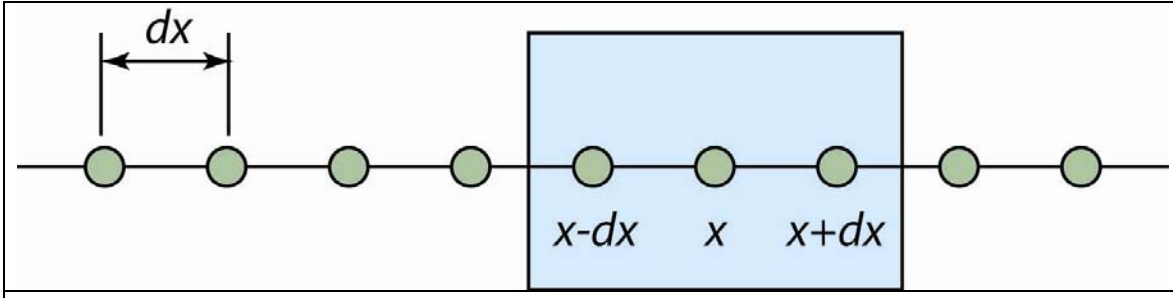
The art of parallel programming is identifying the part of the problem which can be efficiently parallelized. As a quick example let's look at the 1-D wave equation. We can write this as:

$$\frac{\partial^2 p(x,t)}{\partial t^2} = c^2 \frac{\partial^2 p(x,t)}{\partial x^2}$$

Where **p** is pressure and **c** is velocity. Here we have time derivatives that describe how the system evolves with time and spatial derivatives describing the interaction of different particles.

We can solve this equation by a simple finite difference approximation:

$$p(t+dt) = 2\,p(t) - p(t-dt) + \frac{p(x+dx) - 2\,p(x) + p(x-dx)}{dx^2} c^2 dt^2$$



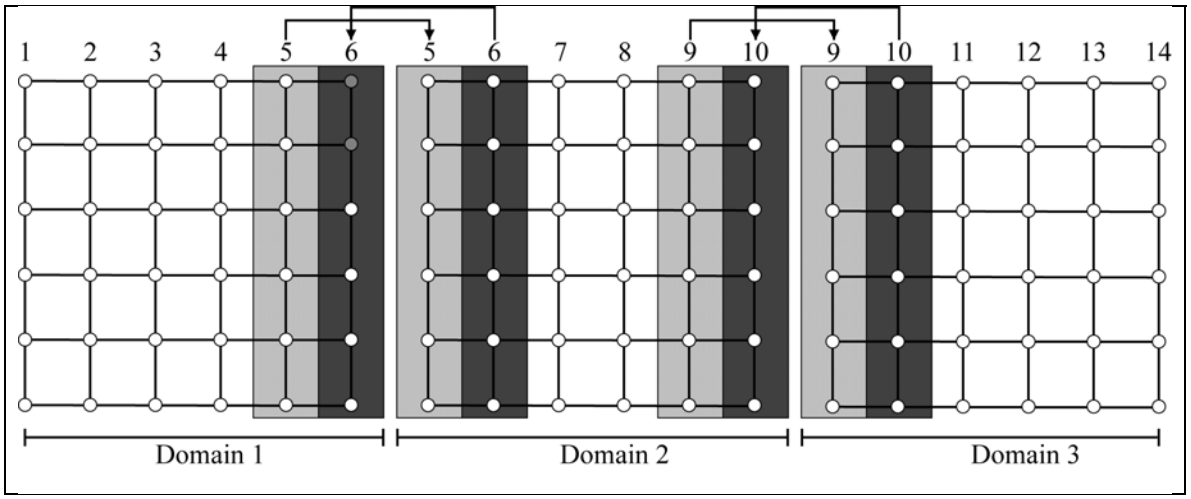
Consider we are solving our wave equation at discrete spatial points represented by the green circles separated by a distance of dx. At the point x, solution of the spatial derivative (2nd derivative in this case) only involves the values of pressure at the points in the immediate vicinity of x (e.g., using a 3-point centered difference approximation the solution only involves the two neighboring points inside the blue box).

Note that what happens in the near future (**t** + **dt**) at some point **x** only depends on:

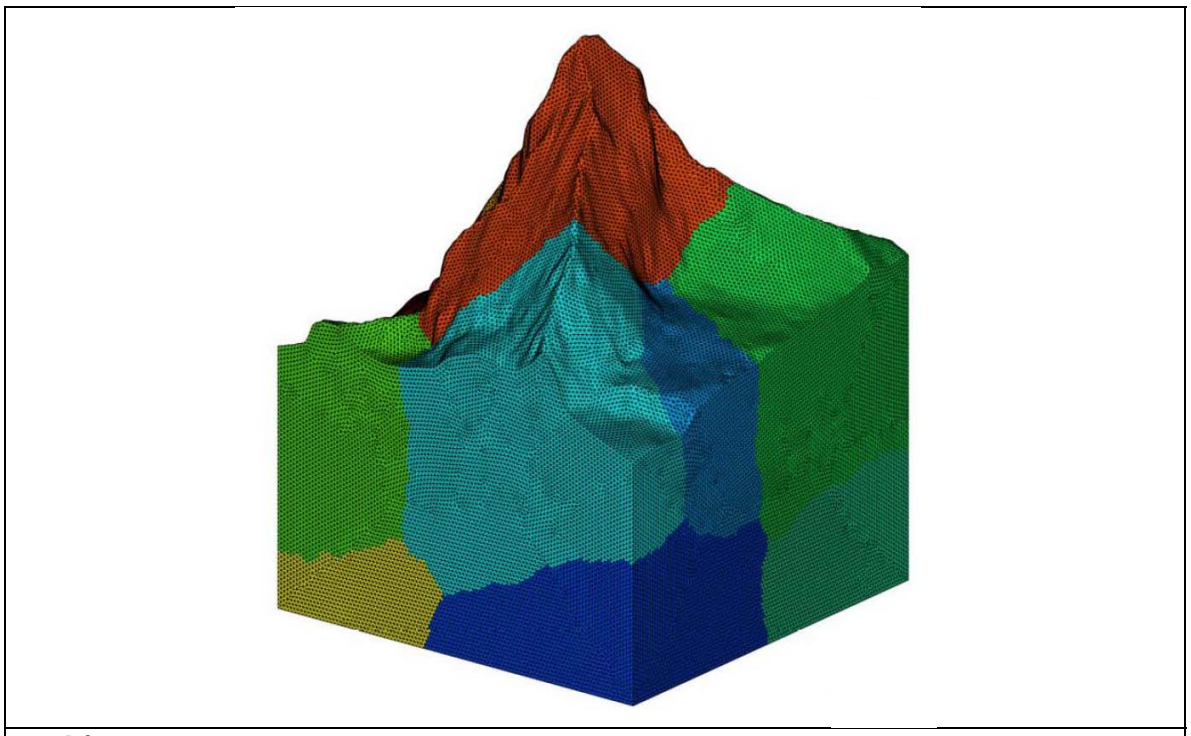
- the present time (**t**),
- the immediate past (**t** – **dt**)
- and the state of the system in the nearest neighborhood of **x** (**x** ± **dx**)

This type of behavior is inherent in physics. The key now is to determine how best to subdivide the problem amongst the many processors you have available to you. That is, we want to **parallelize** the problem. It is important to note our desire is to **Parallelize** and not **Paralyze** our code.

In the example above it makes sense that we may want to divide the problem up spatially and have different processors work on chunks of the problem that are closely located in space. An equivalent 2D example may look as follows, where we have here shown the 2D grid divided up into 3 blocks.



However, these spatial divisions can get much more difficult in 3D problems. Below is an example grid from Martin Käser (Ludwig Maximilians University, Munich) where each color represents the part of the problem that a different node will work on.



Grid from Martin Käser.

One of the primary issues in parallelizing code has to do with the exchanging of information at domain boundaries:

- Each processor is working on a single section of the code, but at the boundaries requires information from other processors. For example, in our example of the 1D wave equation we may need the pressure values being calculated on other processors to be able to calculate the FD approximation in our own domain.

- Hence, some form of communication needs to take place. This is where the Message Passing comes into play.

We have two fundamental concerns: (1) Load balancing – we want to divide the problem up as equally as possible so as to keep all of the processors busy, and (2) we want to minimize the inter-processor communication. There is generally a tradeoff between processing and communication.

3. Parallel Programming Environments

Parallel programming requires special programming techniques to be able to exploit their speed. Typically, Fortran produces faster code than C or C++ (this is because it is really hard to optimize pointers) and as a result most supercomputer applications are written in Fortran. This is definitely the case in Seismology (all major supercomputing codes in global seismology are written in Fortran 90) and appears to be the case in meteorology from the people I've talked to. In any case, parallel programming can be done in either Fortran, C, or C++ (and in other languages as well, but less commonly). When I was employed at the Arctic Region Supercomputing Center I asked one of the people running the center what language was used the most in applications running on their computers. I was actually a little surprised that greater than 90% of the applications were written in Fortran, however this was dominated by the meteorologists who were running the weather models. I don't know if this paradigm is true elsewhere.

How one exploits the parallelism depends on the computing environment. For each environment there are different utilities available:

Distributed Memory:

- **MPI** (**M**essage **P**assing **I**nterface)
- **PVM** (**P**arallel **V**irtual **M**achine)

Shared Memory – Data Parallel (also known as multi-threading):

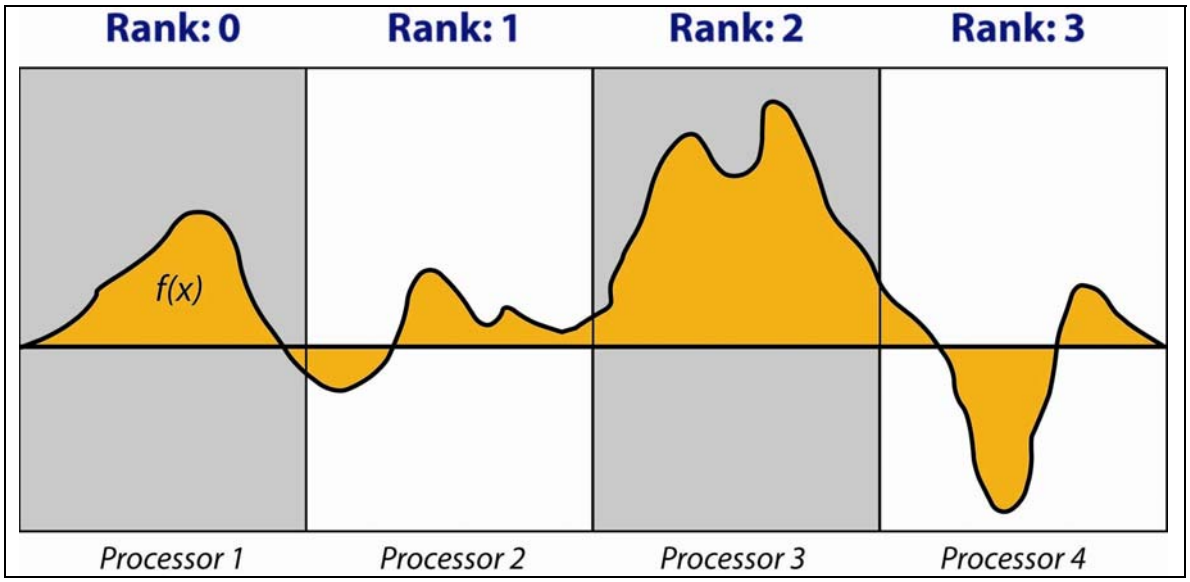
- **OpenMP** (**Open M**ulti-**P**rocessing)
- **Posix Threads** (**P**ortable **O**perating **S**ystem **I**nterface)

Usually parallel computers address all of these environments. It is up to the programmer to decide which one suits the problem best. In this class we will focus on distributed memory systems and **MPI** programming which is the most common. However, it is not uncommon to use a combination of methods. Think about our example of how supercomputers are set up. One node is a shared memory environment, and looking across nodes is a distributed memory environment. Hence, it is common to use **OpenMP** to deal with parallelization between processors on the same node, and to use

MPI to deal with the parallelization across nodes.

5. Intro to Message Passing Concepts

Here we will start to describe the concepts of actually writing parallel code using the Message Passing Interface (MPI). The key point is that we are going to write our code to solve a problem where we have several different processors working on a different chunk of the problem. For example, suppose we are going to **numerically integrate a 2D function**. The first thing we might do is decide how we are going to break this problem up. We might just want each processor to compute an equal part of the integral. Hence, if I have 4 processors at my disposal each processor might try and compute these parts of the integral



The main points here is that:

- I divided my problem up into 4 sections, and have decided that each processor is going to do the numerical integration in each one of these sections.

- In parallel programming we refer to each of our sections as **ranks**, and we start our numbering scheme with **rank = 0**. Hence, we refer to the part of the problem that our first processor is working on as **rank 0**. Our second processor is working on **rank 1**, etc.

- Our task as a programmer is to tell each processor what it should be doing. That is, we specify the actions of a process performing part of the computation rather than the action of the entire code. In this example we are simply telling every processor to sum up an area under the curve, but we are telling each processor to calculate this sum under a different region of the curve.

- Note that each rank is only solving a part of the integral. To determine the final answer we have to **communicate** the result of all ranks to just a single rank and sum the answers.

As another example, imagine that we just have two processors. At the start of the code execution we initialize the variable **X = 0.0**.

	Processor 1	Processor 2
Initialization:	**myrank: 0** **X = 0.0**	**myrank: 1** **X = 0.0**

Here we use the variable **myrank** to tell us which process we are using. At this point we could provide some code. For example:

	Processor 1 Processor 2

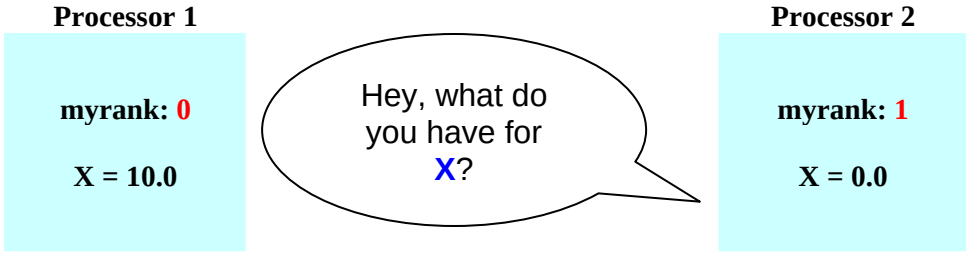
Code:	**IF (myrank == 0) THEN**
	X = X + 10.0
	ENDIF

As you can see our code is giving a specific instruction based on which processor is doing the work. After execution of this line of code we get:

	Processor 1	Processor 2
Result:	**myrank: 0** **X = 10.0**	**myrank: 1** **X = 0.0**

And the important point that although we are just using the single variable **X**, it can take on different values depending on which processor we are referring to.

But, at some point one processor may be interested in what the value of a variable is on another processor. For example, Processor 2 wants to know what **X** is on Processor 1:



To determine this we have to **Pass** a **Message** from **rank 1** to **rank 0** asking it to supply its value of **X**, and then we have to send the answer from **rank 0** back to **rank 1**.

In **Passing Messages** the following items must be considered:

- Which processor is sending a message? (**which rank**)
- Where is the data on the sending processor? (**which variable**)
- What kind of data is being sent? (**e.g., integer, real, …**)

- How much data is being sent? (**e.g., a single integer, how many array elements**)
- Which processor(s) is (are) receiving the message? (**which rank**)
- Where should the data be left on the receiving processor? (**which variable**)
- How much data is the receiving processor prepared to accept? (**e.g., how many array elements**)

In the next lecture we will show the details of how this is done using the Message Passing Interface.

6. Homework

This is a buy week. Have fun!