Why Parallelism/High-performance?

- High performance is always a relative term. Algorithmic ambition vs. Affordable systems is the main tradeoff.

- Need for larger problems, e.g., more points in a discretization of a continuous model — improves fidelity to given model but does not improve the model; it may or may not change the algorithm.

- Need for larger problems, e.g., more points in a sampled image — improves fidelity to observed scenario; it may or may not change the algorithm.
• Need for more sophisticated models, e.g., more sophisticated physics in an atmospheric model – increases computation, tends to introduce inhomogeneity, hopefully improves science.

• Need for more sophisticated algorithms, e.g., improved communications/signal processing – improves capabilities but may have too large of a cost for current processing systems.
Issues from 4 different areas must be considered when discussing parallel or high-performance processing

- Architecture

- Software

- Algorithms and Applications

- Performance Models and Metrics
Architecture

• Key features of architecture must be identified along the performance improving assumptions upon which that are based.

• The limitations of these assumptions must be identified.

• Three main components to consider:
  – processor design
  – memory system design
  – network design
Software

- The levels of software / architecture interaction available and those required to achieve high performance must be identified (they are not always consistent).

- Low-level SW support (assembler level) tends to be used to make explicit control of specialized features available.

- Higher-level SW support (Fortran) tries to compromise with more generic constructs that can be mapped to low-level SW in different ways on different architectures.
• This can be carried to the extreme of masking the details of the architecture completely and using a programming paradigm that contradicts the underlying architectural assumptions — this is typically done in the name of portability.
- Explicit vs. Implicit parallel programming

- Even though low and higher-level software constructs are available on the machine, they may not be available to the generic user.

- Often systems require the use of support tools such as restructuring compilers to go from generic sequential code to parallel code that is seen and compiled only by the system software and not the user.

- Performance monitoring tools are critical to tuning the performance of any code. They can also come in explicit vs. implicit forms.
Algorithms and Applications

- The ultimate measure of success is given by the algorithm/application community.

- While theoretical and aesthetic beauty in parallel algorithms are satisfying – fast, accurate and robust algorithms solving real application problems are better.

- Merging architecture and software considerations with algorithm and application insights is the most difficult part of parallel programming.

- The use of legacy code makes this extremely difficult – avoid it if possible.
Performance Models and Metrics

- The interaction of the architectural/software parameters and trends with algorithmic parameters must be expressed in some tractable fashion to guide algorithm design and tuning.

- Performance models – typically combining empirical and theoretical forms – accomplish this.

- Typically a tradeoff exists between level of accuracy and cost of evaluation for these model, i.e., the most accurate model of an algorithm on an architecture is to actually implement both and time it. This does tend to be expensive.
• Some simplification is required for the sake of tractability.

• Performance metrics that emphasize what is considered acceptable performance as a function of architectural and algorithmic parameters are also required to assess progress and declare victory or defeat.
Rules of Thumb for Efficient Processing

1. Do not wait until one thing is finished to start something else.

2. Do not do more work than you must.

3. Do not let the cost of following Rule 1 or Rule 2 outweigh the benefit of following Rule 1 or Rule 2.
Some Important Considerations

1. Correctness – Efficient execution should not destroy the correctness of the algorithm. (Sometimes this is not easy to test.)

2. Accuracy – Efficient execution may alter stability and roundoff properties of numerical algorithms compared to slower and more reliable algorithms. (This can be very difficult to check. There are many computationally optimal algorithms that have suspect stability.) Parallelism can also affect the quality of the solution of nonnumerical algorithms due to altered search paths.

3. Time – If it takes longer to figure out that you can save work than the time it takes to do the work you save then forget it.

4. Architecture (HW and SW) – Be careful that the technique you are considering maps well to the HW and SW support of the machine.
Considerations (cont.)

5. Generality – It is very easy to build a very fast program or machine that solves one problem for one set of data; it is not however very useful so to do.

6. Portability – The classic dilemma for high-performance algorithm and library designers. To get a substantial portion of the peak possible for a machine or more precisely for a machine running the best possible implementation of your code you have to customize for the mix. The resulting code will not achieve the same performance percentage on another machine that it significantly different in SW or HW architecture.
• We have seen two granularities of parallelism already within a single processor
  - pipelining – extremely very fine granularity, i.e., multiple sub-operations running simultaneously
  - functional unit parallelism – very fine grain, i.e., multiple machine instructions running simultaneously

• multiple processors in a parallel processing system allow the use of more aggressive parallelism
  - fine grain – tens of instructions executed by each processor as basic work unit, e.g., a single iteration of a loop
  - coarse grain – hundreds to thousands of instructions executed by each processor
as basic work unit – more complicated functions or subroutines that make up a significant portion of the user’s job

• multiple parallel processors combined to produce a larger system – clustered computing – tends to exploit many levels of granularity
Characterizing Systems

- Two key axes

- programming paradigm
  - shared memory
  - distributed memory
  - data parallel (hybrid of the two)

- architectural paradigm
  - relative position of processors and memory banks
  - network topology
  - communication/coherence support
  - control(scheduling)/synchronization support
Programming Paradigm

- A user’s job consists of multiple processes that must coordinate with each other in order to complete correctly.

- For simplicity, it is assumed that there is one process per processor (this is not always true in practice)

- the function of the process relative to the high-level code the user provides depends on the programming paradigm and code constructs used

- shared memory and distributed memory programming are the two main approaches

- they are often used simultaneously on certain types of systems
Shared Memory

- all processes share the same address space, i.e., if process 1 and process 2 both ask for the contents of address 15 they get the same information

- communication between processes is implicit, i.e., via load and stores to shared memory

- control(scheduling)/synchronization is explicit, i.e., deciding when and where operations are performed in order to guarantee correctness must be specified at some level of coding

- private process variables are supported in software
• parallel constructs range from loops to large grain tasks

• loops imply worker processes on each processor that consume iterations, large grain tasks are usually given explicitly by user code
Distributed Memory

- each process has a distinct address space, i.e., if process 1 and process 2 both ask for the contents of address 15 they access two different physical locations each of which is in their private address space (and typically in their private physical memory bank)

- communication between processes is explicit, i.e., messages are sent and received by processes

- control(scheduling)/synchronization is implicit, i.e., operations can only be performed when all of the required data is present in the private memory of the computing process so synchronization follows from the data be sent and received
• shared variables do not exist in any way

• typically user code specifies a task that is sent to each processor and that determines what it is to do based on its process number
Architectural Paradigm

- the relative placement of the processors and memory banks in the system determines is the basic attribute of the memory system

- **physically centralized memory** places processors on one side of the network and memory banks on the other

- such a system has uniform memory access time, i.e., the cost to access all locations in physical memory (not in cache) from a processor is the same
• **physically distributed memory** places a processor and memory bank in a pair that communicates with other pairs via the network

• since the processor has a distinct and faster path to the memory bank in its pair this system has a nonuniform memory access time
• the communication support is the next key aspect of the memory system

• **physically shared memory** means that a processor can issue an address and the network and memory banks can supply the associated data **from any physical location in the entire memory system**

• therefore, no processor other than the one issuing the request needs to be involved when data is accessed (read or written)
• **physically private memory** means that an address issued from a processor is only interpreted in the memory bank in the same pair as the processor. If a processor wants data located in the memory bank in another pair it must receive the data in a message sent by the processor in the pair with the memory bank that contains the data.

• the processor that **owns** the data must be involved by explicitly sending the data to the processor that needs it.
Caches and Parallel Processors

• if each processor has a cache then potentially there can be $p + 1$ copies of an addresses data in the a system with $p$ processors

• on a physically shared memory system the system may or may not guarantee that a processor requesting address A will get the latest contents of that address

• if the system simply goes to address A in physical memory and not check caches for more recently updated versions of the data it may not get the latest version and is called incoherent physically shared memory
• for such systems the programming paradigm is often restricted more than pure shared memory for nonexpert users, e.g., data parallel is often used

• expert users are required to program incoherent shared memory and can often produce very efficient code

• if the system supports automatically finding the latest copy of the contents of address A then the system is coherent physically shared memory

• mechanisms to support this will be discussed later