Researchers Give Update on Road to Parallelism

EE Times (03/19/10) Merritt, Rick

University of Illinois researchers have taken several small steps toward developing new parallel programming models to tap the many-core processors of the future. The DeNovo project attempts to define a new and more rigorous way of utilizing shared memory. It is working concurrently with a separate effort to define a deterministic, parallel language primarily based on a parallel version of Java and eventually migrating to a parallel version of C++.

The chip project that is nearest to testing is the 1,024-core Rigel processor architecture targeting high density, high throughput computing, which would be programmed through a task-level applications programming interface aimed at chores in imaging, computer vision, physics, and simulations. The Bulk Architecture chip design is testing the notion of atomic transactions.
Titan has AMD Opteron CPUs in conjunction with Nvidia Tesla GPUs to improve energy efficiency while providing an order of magnitude increase in computational power over Jaguar. It utilizes 18,688 CPUs paired with an equal number of GPUs to perform at a theoretical peak of 27 petaFLOPS; however, in the LINPACK benchmark used to rank supercomputers’ speed, it performed at 17.59 petaFLOPS. This was enough to take first place in the November 2012 list by the TOP500 organisation.
Computer performance

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbr.</th>
<th>FLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>kiloFLOPS</td>
<td>kFLOPS</td>
<td>$10^9$</td>
</tr>
<tr>
<td>megaFLOPS</td>
<td>MFLOPS</td>
<td>$10^9$</td>
</tr>
<tr>
<td>gigaFLOPS</td>
<td>GFLOPS</td>
<td>$10^9$</td>
</tr>
<tr>
<td>teraFLOPS</td>
<td>TFLOPS</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>petaFLOPS</td>
<td>PFLOPS</td>
<td>$10^{15}$</td>
</tr>
<tr>
<td>exaFLOPS</td>
<td>EFLOPS</td>
<td>$10^{18}$</td>
</tr>
<tr>
<td>zettaFLOPS</td>
<td>ZFLOPS</td>
<td>$10^{21}$</td>
</tr>
<tr>
<td>yottaFLOPS</td>
<td>YFLOPS</td>
<td>$10^{24}$</td>
</tr>
</tbody>
</table>

Instruction-level parallelism

A canonical five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

Instructions can be grouped together only if there is no data dependency between them.
TÜ HPC – v1, Feb 2, 2009

- 42 nodes, 2x4-core = 336 core
- 32GB RAM / node
- Infiniband fast interconnect
- http://www.hpc.ut.ee/
- Job scheduling (Torque)
- MPI

---

Grace Hopper - Nanosecords
http://www.youtube.com/watch?v=JEpSkWZj8

Latency
The single data rate switch chips have a latency of 200 nanoseconds, and DDR switch chips have a latency of 140 nanoseconds. The end-to-end latency range is from 1.07 microseconds MPI latency to 1.26 microseconds. Compare:
The speed of a normal ethernet ping/pong (request and response) is roughly 350us (microseconds) or about .35 milliseconds or .00035 seconds.

---

Single large-memory machines @TartuUniversity

- 80-core, 256GB RAM
- HPC: 512GB,
- EXCS: 160-core cpu, 1TB RAM fall 2011
- >3000 cores now
- 2TB RAM machine

- http://hpc.ut.ee/ for more info!

---

Home, 3y old laptop, wifi
Drivers for parallel computing

- Multi-core processors (2, 4, ... , 64 ... )
- Computer clusters (and NUMA)
- Specialised computers (e.g. vector processors, massively parallel, ...)
- Distributed computing: GRID, cloud, ...
- Need to create computer systems from cheaper (and weaker) components, but many ...
- **One of the major challenges of modern IT**

Slides based on materials from

- Joe Davey
- Matt Maul
- Ashok Srinivasan
- Edward Chrzanski
- Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar
- Dr. Amitava Datta and Prof. Dr. Thomas Ottmann
- http://en.wikipedia.org/wiki/Parallel_computing
- ... and many others

**books**

- CLRS: (3rd edition) pp 772-812 (Chapter 27)

What is a Parallel Algorithm?

- Imagine you needed to find a lost child in the woods.
- Even in a small area, searching by yourself would be very time consuming
- Now if you gathered some friends and family to help you, you could cover the woods in much faster manner...
Terms (CLRS)

- work, span

Parallel complexity

- Schedule on p processors
- schedule depth – max /longest path in schedule/
- N – set of nodes on DAG
- time ti – assigned for each DAG node
- $T_p(n) = \min\{ \max_{i \in N} t_i \}$
- Length of the longest (critical) path!

Boolean circuits, bit-level parallelism

- (Micro)processors
- Basic building blocks,
- bit-level operations
- combine into word-level operations

![NOR - Top view](image)

This is the NOR gate as seen from above.

![8-bit adder](image)
Most of the available algorithms to compute $P_i$, on the other hand, cannot be easily split up into parallel portions. They require the results from a preceding step to effectively carry on with the next step. Such problems are called inherently serial problems.

---

**Time complexity**

- **Parallel computation**
  - A group of processors work together to solve a problem
  - Time required for the computation is the period from when the first processor starts working until when the last processor stops

- **Notation**
  - $P$: Number of processors
  - $T_1$: Time on one processor
  - $T_P$: Time on $P$ processors

**Other terminology**

- **Speedup**: $S = T_1 / T_P$
- **Efficiency**: $E = S / P$
- **Work**: $W = P T_P$
- **Scalability**
  - How does $T_1$ decrease as we increase $P$ to solve the same problem?
  - How should the problem size increase with $P$, to keep $E$ constant?
• Sometimes a speedup of more than N when using N processors is observed in parallel computing, which is called super linear speedup. Super linear speedup rarely happens and often confuses beginners, who believe the theoretical maximum speedup should be N when N processors are used.

• One possible reason for a super linear speedup is the cache effect.

• Super linear speedups can also occur when performing backtracking in parallel: One thread can prune a branch of the exhaustive search that another thread would have taken otherwise.

Amdahl law
is used to find the maximum expected improvement to an overall system when only part of the system is improved.

Amdahl’s Law

• e.g. 5% non parallelizable => speedup can not be more than 20x !

• http://en.wikipedia.org/wiki/Amdahl%27s_law

Sequential programme

Communications

• Simple model for analyzing algorithm communications performance:

\[
\text{tcomm} = \text{communication time} \\
\text{tstartup} = \text{#words} \times \text{tdata} \\
\text{latency} = \text{#words} \times 1/(\text{words/second}) \\
\text{latency} = a + w*()} \\
\text{latency and bandwidth}
I/O model

- We will ignore I/O issues, for the most part
- We will assume that input and output are distributed across the processors in a manner of our choosing

Example: Sorting
- Input: $x_1, x_2, ..., x_n$
  - Initially, $x_i$ is on processor $i$
  - Output: $x_{i_1}, x_{i_2}, ..., x_{i_n}$
- $x_i$ on processor $i$
- $x_i < x_{i+1}$

Important points

- **Efficiency**
  - Increases with increase in problem size
  - Decreases with increase in number of processors
- **Aggregation of tasks to increase granularity**
  - Reduces communication overhead
- **Data distribution**
  - 2-dimensional may be more scalable than 1-dimensional
  - Has an effect on load balance too
- **General techniques**
  - Divide and conquer
  - Pipelining

Parallel Architectures

- **Single Instruction Stream, Multiple Data Stream (SIMD)**
  - One global control unit connected to each processor
- **Multiple Instruction Stream, Multiple Data Stream (MIMD)**
  - Each processor has a local control unit

Parallel Computing Architectures

**Flynn’s Taxonomy**

<table>
<thead>
<tr>
<th></th>
<th>Single Instruction Stream</th>
<th>Multiple Instruction Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data Stream</td>
<td>SISD uniprocessors</td>
<td>SIMD Processor arrays</td>
</tr>
<tr>
<td>Multiple Data Stream</td>
<td></td>
<td>MIMD Multiprocessors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multicomputers</td>
</tr>
</tbody>
</table>

Architecture (continued)

- **Shared-Address-Space**
  - Each processor has access to main memory
  - Processors may be given a small private memory for local variables
- **Message-Passing**
  - Each processor is given its own block of memory
  - Processors communicate by passing messages directly instead of modifying memory locations
Parallel Computing Architectures

<table>
<thead>
<tr>
<th>Memory Model</th>
<th>Shared Address Space</th>
<th>Individual Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized memory</td>
<td>SMP (Symmetric Multiprocessor)</td>
<td>N/A</td>
</tr>
<tr>
<td>Distributed memory</td>
<td>NUMA (Non-Uniform Memory Access)</td>
<td>MPP (Massively Parallel Processors)</td>
</tr>
</tbody>
</table>

**NUMA architecture**

![NUMA architecture diagram](image)

**SMP - Symmetric multiprocessing**

- **Shared Memory**
  - All processes share the same address space
  - Easy to program, also easy to program poorly
  - Performance is hardware dependent; limited memory bandwidth can create contention for memory

- **MIMD (multiple instruction multiple data)**
  - Each parallel computing unit has an instruction thread
  - Each processor has local memory
  - Processors share data by message passing
  - Synchronization must be explicitly programmed into a code

- **NUMA (non-uniform memory access)**
  - Distributed memory in hardware, shared memory in software, with hardware assistance (for performance)
  - Better scalability than SMP but not as good as a full distributed memory architecture

**MPP – Massively Parallel**

- Each node is an independent system having its own:
  - Physical memory
  - Address space
  - Local disk and network connections
  - Operating system

**Related Reading**

- [http://www.theregister.co.uk/2013/05/01/amd_huma/](http://www.theregister.co.uk/2013/05/01/amd_huma/)

---

![AMD reveals potent parallel processing breakthrough](image)
MPP

• Short for **massively parallel processing**, a type of computing that uses many separate CPUs running in parallel to execute a single program. MPP is similar to **symmetric processing** (SMP), with the main difference being that in SMP systems all the CPUs share the same memory, whereas in MPP systems, each CPU has its own memory. MPP systems are therefore more difficult to program because the application must be divided in such a way that all the executing segments can communicate with each other. On the other hand, MPP don't suffer from the bottleneck problems inherent in SMP systems when all the CPUs attempt to access the same memory at once.

Interconnection Networks

• Static
  – Each processor is hard-wired to every other processor

• Dynamic
  – Processors are connected to a series of switches

Why Do Parallel Computing

• Time: Reduce the turnaround time of applications
• Performance: Parallel computing is the only way to extend performance toward the TFLOP realm
• Cost/Performance: Traditional vector computers become too expensive as one pushes the performance barrier
• Memory: Applications often require memory that goes beyond that addressable by a single processor
Cont...

• Whole classes of important algorithms are ideal for parallel execution. Most algorithms can benefit from parallel processing such as Laplace equation, Monte Carlo, FFT (signal processing), image processing
• Life itself is a set of concurrent processes
  — Scientists use modelling so why not model systems in a way closer to nature

Some Misconceptions

• Requires new parallel languages?
  — No. Uses standard languages and compilers (Fortran, C, C++, Java, Occam)
  • However, there are some specific parallel languages such as Qlisp, MuT and others – check out: http://ceu.fi.udc.es/SAL/C/1/index.shtml

• Requires new code?
  — No. Most existing code can be used. Many production installations use the same code base for serial and parallel code.

Cont...

• Requires confusing parallel extensions?
  — No. They are not that bad. Depends on how complex you want to make it. From nothing at all (letting the compiler do the parallelism) to installing semaphores yourself
• Parallel computing is difficult:
  — No. Just different and subtle. Can be akin to assembler language programming 😊

The PRAM Model

• **Parallel Random Access Machine**
  — Theoretical model for parallel machines
  • p processors with uniform access to a large memory bank
  • MIMD
  • UMA (uniform memory access) – Equal memory access time for any processor to any address

Memory Protocols

• Exclusive-Read Exclusive-Write
• Exclusive-Read Concurrent-Write
• Concurrent-Read Exclusive-Write
• Concurrent-Read Concurrent-Write

• If concurrent write is allowed we must decide which value to accept
**Linearizability**

In concurrent programming, an operation (or set of operations) is **atomic**, **linearizable**, **indivisible** or **uninterruptible** if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from concurrent processes. Additionally, atomic operations commonly have a **succeed-or-fail** definition—they either successfully change the state of the system, or have no apparent effect.

Atomicity is commonly enforced by **mutual exclusion**, whether at the hardware level building on a cache coherence protocol, or the software level using semaphores or locks. Thus, an atomic operation does not actually occur instantaneously. The benefit comes from the appearance the system behaves as if each operation occurred instantly, separated by pauses. Because of this, implementation details may be ignored by the user, except insofar as they affect performance. If an operation is not atomic, the user will also need to understand and cope with sporadic extraneous behaviour caused by interactions between concurrent operations, which by their nature are likely to be hard to reproduce and debug.

An object (built in hardware or software) is **linearizable** if its operations seem to occur instantaneously.

More formally:

For every execution that uses the objects, we can choose a \( x \) inside each operation's time interval so that all operations would return the same results if they were performed sequentially in the order of their \( x \).

The \( x \) is called the **linearization point** of the operation.

**https://en.wikipedia.org/wiki/Linearizability**
Primitive level operations

- Processors have instructions that can be used to implement locking and lock-free and wait-free algorithms. The ability to temporarily inhibit interrupts, ensuring that the currently running process cannot be context switched, also suffices on a uniprocessor. These instructions are used directly by compiler and operating system writers but are also abstracted and exposed as bytecodes and library functions in higher-level languages:

- Locking – coarse or fine-grained
  - Non-blocking (some can always advance)
  - Wait-free (each thread will eventually finish)
- Update value (if the same)
- ABA problem
- Update counter
- CAS (compare and swap)
- Lock-free data structures...
- Slides from Eric Ruppert

Non-blocking algorithm

In computer science, an algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread. For some operations, these algorithms provide a useful alternative to traditional blocking implementations. A non-blocking algorithm is lock-free if there is guaranteed system-wide progress, and wait-free if there is also guaranteed per-thread progress.

Example: Merge Sort

0 1 2 3 4 5 6 7
P1  P2  P3  P4

0 1 2 3 4 5 6 7
P1  P2

0 1 2 3 4 5 6 7
P1
Merge Sort Analysis

- Number of compares
  \[
  1 + 3 + \ldots + (2^{i-1}) + \ldots + (n-1) \\
  \sum_{i=1}^{\log_2 n} 2^i - 1 = 2n - 2\log_2 n = \Theta(n)
  \]
- We have improved from \(n\log(n)\) to \(n\) simply by applying the old algorithm to parallel computing, by altering the algorithm we can further improve merge sort to \((\log n)^2\)

CLRS

- parallel (to execute for-loop iterations in parallel)
- spawn (create a new task/thread), and
- sync (wait for all spawned children)

\[
\begin{align*}
Fib(n) & = \begin{cases}
  1 & \text{if } n = 1 \\
  2 & \text{if } n = 2 \\
  3 & \text{return } n \\
  4 & \text{else}
\end{cases} \\
& \quad \begin{cases}
  \text{x = } Fib(n-1) \\
  \text{y = } Fib(n-2) \\
  \text{return } x+y
\end{cases}
\]

Parallel Design and Dynamic Programming

- deterministic – always the same result
- non-deterministic – may depend on order, runto run
- race

\[
\begin{align*}
\text{Race-Example()} & \quad \text{x=0} \\
& \quad \text{parallel for } i=1 \text{ to } 2 \\
& \quad \quad \text{x = x+1} \\
& \quad \text{print } x
\end{align*}
\]
Current Applications that Utilize Parallel Architectures

- Computer Graphics Processing
- Video Encoding
- Accurate weather forecasting
- Scientific computing, modelling
- ...

Parallel addition features

- If \( n \gg P \)
  - Each processor adds \( n/P \) distinct numbers
  - Perform parallel reduction on \( P \) numbers
  - \( T_P \approx n/P + (1 + t_b + t_s) \log P \)
  - Optimal \( P \) obtained by differentiating wrt \( P \)
    - \( P_{opt} \approx n/(1 + t_s + t_b) \)
  - If communication cost is high, then fewer processors ought to be used
    - \( E = [1 + (1 + t_s + t_b) P \log P/n]^2 \)

- As problem size increases, efficiency increases
- As number of processors increases, efficiency decreases

Some common collective operations

- Broadcast
- Gather
- Scatter
- All Gather

Broadcast

- \( T \approx (t_s + L_b) \log P \)
- \( L \): Length of data

Gather/Scatter

- Gather: Data move towards the root
- Scatter: Review question
- \( T \approx t_s \log P + PL_b \)

All gather

- Equivalent to each processor broadcasting to all the processors

Note: \( 2n^{\log_2 P-1}/2 = 2n^{\log_2 P-1}/(2-1) = P-1 \)
All gather

\[ T_n \sim \log P + PL_b \]

Matrix-vector multiplication

- \( c = A b \)
  - Often performed repeatedly
  - \( b_i = A b_{i-1} \)
  - We need same data distribution for \( c \) and \( b \)
- One dimensional decomposition
  - Example: row-wise block striped for \( A \)
    - \( b \) and \( c \) replicated
    - Each process computes its components of \( c \) independently
    - Then all-gather the components of \( c \)

1-D matrix-vector multiplication

- Each process computes its components of \( c \) independently
  - Time = \( \Theta(n^2/P) \)
- Then all-gather the components of \( c \)
  - Time = \( t_s \log P + t_b n \)
- Note: \( P < n \)

2-D matrix-vector multiplication

- Processes \( P_a \) sends \( B_i \) to \( P_b \)
  - Time: \( t_s + \log P \)
- Processes \( P_b \) broadcast \( B_i \) to all \( P_i \)
  - Time = \( t_b n \log P / P + t_s \log P \)
- Processes \( P_i \) compute \( C_j = A B_i \)
  - Time = \( \log P \)
- Processes \( P_i \) reduce \( C_j \) on to \( P_{ij} \), \( 0 \leq i < P \)
  - Time = \( t_b \log P / P + t_s n \log P / P^{0.5} \)
- Total time = \( \Theta(n^2/P + t_s \log P + t_b n \log P / P^{0.5}) \)
  - \( P \leq n \)
  - More scalable than 1-dimensional decomposition
• 31.2 Strassen’s algorithm for matrix multiplication

This section presents Strassen’s remarkable recursive algorithm for multiplying \( n \times n \) matrices that runs in \( (n^{\log_2 7}) = O(n^{2.81}) \) time. For sufficiently large \( n \), therefore, it outperforms the naïve \( (n^3) \) matrix-multiplication algorithm MATRIX-MULTIPLY from Section 26.1.

\[
C = AB
\]

\[
\begin{pmatrix}
  r & s \\
  t & u
\end{pmatrix} = \begin{pmatrix}
  a & b \\
  c & d
\end{pmatrix} \begin{pmatrix}
  e & f \\
  g & h
\end{pmatrix}
\]

- \( r = ae + bf \)
- \( s = ag + bh \)
- \( t = ce + df \)
- \( u = cg + dh \)
- \( T(n) = 8T(n/2) + \Theta(n^2). = O(n^3) \)

Matrix operations are parallelizable

- Problems that are expressed in forms of matrix operations are often easy to automatically parallelise
- Fortran, etc – programming languages can achieve that at no extra effort

The Process

- A running executable of a (compiled and linked) program written in a standard sequential language (i.e. F77 or C) with library calls to implement the message passing
- A process executes on a processor
- All processes are assigned to processors in a one-to-one mapping (simplest model of parallel programming)
- Other processes may execute on other processors
- A process communicates and synchronizes with other processes via messages
- A process is uniquely identified by:
  - The node on which it is running
  - Its process id (PID)
- A process does not migrate from node to node (though it is possible for it to migrate from one processor to another within a SMP node).
Solving Problems in Parallel

It is true that the hardware defines the parallel computer. However, it is the software that makes it usable.
Parallel programmers have the same concern as any other programmer:
- Algorithm design,
- Efficiency
- Debugging ease
- Code reuse, and
- Lifecycle.

Cont.

However, they are also concerned with:
- Concurrency and communication
- Need for speed (nee high performance), and
- Plethora and diversity of architecture

Fosters Four step Process for Designing Parallel Algorithms

1. Partitioning – process of dividing the computation and the data into many small pieces – decomposition
2. Communication – local and global (called overhead) minimizing parallel overhead is an important goal and the following check list should help the communication structure of the algorithm
   1. The communication operations are balanced among tasks
   2. Each task communicates with only a small number of neighbours
   3. Tasks can perform their communications concurrently
   4. Tasks can perform their computations concurrently

Cont...

3. Agglomeration is the process of grouping tasks into larger tasks in order to improve the performance or simplify programming. Often in using MPI this is one task per processor.
4. Mapping is the process of assigning tasks to processors with the goal to maximize processor utilization

Solving Problems in Parallel

- Decomposition determines:
  - Data structures
  - Communication topology
  - Communication protocols
- Must be looked at early in the process of application development
- Standard approaches

Decomposition methods

- Perfectly parallel
- Domain
- Control
- Object-oriented
- Hybrid/layered (multiple uses of the above)
For the program

- Choose a decomposition
  - Perfectly parallel, domain, control etc.
- Map the decomposition to the processors
  - Ignore topology of the system interconnect
  - Use natural topology of the problem
- Define the inter-process communication protocol
  - Specify the different types of messages which need to be sent
  - See if standard libraries efficiently support the proposed message patterns

Perfectly parallel

- Applications that require little or no inter-processor communication when running in parallel
- Easiest type of problem to decompose
- Results in nearly perfect speed-up

Domain decomposition

- In simulation and modelling this is the most common solution
  - The solution space (which often corresponds to the real space) is divided up among the processors. Each processor solves its own little piece
  - Finite-difference methods and finite-element methods lend themselves well to this approach
  - The method of solution often leads naturally to a set of simultaneous equations that can be solved by parallel matrix solvers
  - Sometimes the solution involves some kind of transformation of variables (i.e. Fourier Transform). Here the domain is some kind of phase space. The solution and the various transformations involved can be parallelized

Output Data Decomposition: Example

A partitioning of output data does not result in a unique decomposition into tasks. For example, for the same problem as in previous foil, with identical output data distribution, we can derive the following two (other) decompositions:

<table>
<thead>
<tr>
<th>Decomposition I</th>
<th>Decomposition II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1: ( C_{1,1} = A_{1,1} B_{1,1} )</td>
<td>Task 1: ( C_{1,1} = A_{1,1} B_{1,1} )</td>
</tr>
<tr>
<td>Task 2: ( C_{1,1} = A_{1,1} + A_{1,2} B_{2,1} )</td>
<td>Task 2: ( C_{1,1} = C_{1,1} + A_{1,2} B_{2,1} )</td>
</tr>
<tr>
<td>Task 3: ( C_{1,2} = A_{1,1} B_{1,2} )</td>
<td>Task 3: ( C_{1,2} = A_{1,1} B_{1,2} )</td>
</tr>
<tr>
<td>Task 4: ( C_{1,2} = C_{1,2} + A_{1,2} B_{2,2} )</td>
<td>Task 4: ( C_{1,2} = C_{1,2} + A_{1,2} B_{2,2} )</td>
</tr>
<tr>
<td>Task 5: ( C_{1,1} = A_{1,1} B_{1,1} )</td>
<td>Task 5: ( C_{1,1} = A_{1,1} B_{1,1} )</td>
</tr>
<tr>
<td>Task 6: ( C_{1,2} = A_{1,1} + A_{1,2} B_{2,1} )</td>
<td>Task 6: ( C_{1,2} = C_{1,2} + A_{1,2} B_{2,1} )</td>
</tr>
<tr>
<td>Task 7: ( C_{2,1} = A_{2,1} B_{1,2} )</td>
<td>Task 7: ( C_{2,1} = A_{2,1} B_{1,2} )</td>
</tr>
<tr>
<td>Task 8: ( C_{2,2} = C_{2,2} + A_{2,2} B_{2,2} )</td>
<td>Task 8: ( C_{2,2} = C_{2,2} + A_{2,2} B_{2,2} )</td>
</tr>
</tbody>
</table>

Control decomposition

- If you cannot find a good domain to decompose, your problem might lend itself to control decomposition
  - Good for:
    - Unpredictable workloads
    - Problems with no convenient static structures
    - One set of control decomposition is functional decomposition
      - Problem is viewed as a set of operations. It is among operators where parallelization is done
      - Many examples in industrial engineering (i.e. modelling an assembly line, a chemical plant, etc.)
      - Many examples in data processing where a series of operations is performed on a continuous stream of data
Control is distributed, usually with some distribution of data structures. Some processes may be dedicated to achieve better load balance.

Examples:
- Image processing: given a series of raw images, perform a series of transformations that yield a final enhanced image. Solve this in a functional decomposition (each process represents a different function in the problem) using data pipelining.
- Game playing: games feature an irregular search space. One possible move may lead to a rich set of possible subsequent moves to search.
  - Need an approach where work can be dynamically assigned to improve load balancing.
  - May need to assign multiple processes to work on a particularly promising lead.

Any problem that involves search (or computations) whose scope cannot be determined a priori are candidates for control decomposition.

Calculations involving multiple levels of recursion (i.e., genetic algorithms, simulated annealing, artificial intelligence).

Discrete phenomena in an otherwise regular medium (i.e., modeling localized storms within a weather model).

Design: rule checking in micro-electronic circuits.

Simulation of complex systems.

Design: game playing, music composing, etc..

Object-oriented decomposition

Object-oriented decomposition is really a combination of functional and domain decomposition.

- Rather than thinking about a dividing data or functionality, we look at the objects in the problem.
- The object can be decomposed as a set of data structures plus the procedures that act on those data structures.
- The goal of object-oriented parallel programming is distributed objects.

Although conceptually clear, in practice it can be difficult to achieve good load balancing among the objects without a great deal of fine tuning.

Works best for the divided problems and in environments where having work ready at the call is more important than worrying about under-worked processors (i.e., a control environment).

Message passing is still explicit (no standard C++ compiler automatically parallelizes over objects).

Summary of Software

- **Compilers**
  - Portable (C++ code)
  - Not parallelized
  - Standard C++ compiler available

- **G++ (GNU)**
  - Good quality implementation exists on the platform.
  - Not scalable
  - Good in predicting parallelism

- **HPF**
  - Good quality implementation exists on the platform.
  - Scalable parallelism

- **MPI**
  - Needs some type of message passing platform.
  - Suitable for small to medium size parallelism.
  - Scalloable speedup studies are reported.
  - Various high-level features are supported.
  - Can adapt to various environments.

- **PThreads**
  - Not recommended
  - Difficult to correct and maintain programs
  - Not scalable to large number of processors

- **POOMA and HPC++**
  - Library is available and it addresses a specific problem.