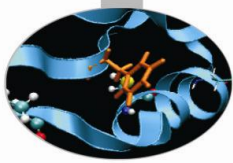
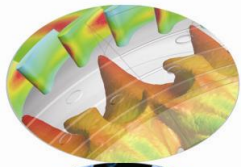
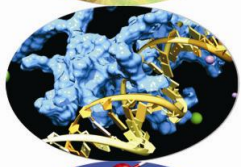
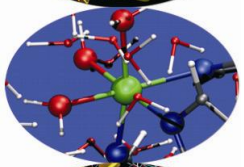
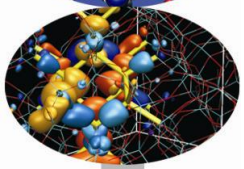
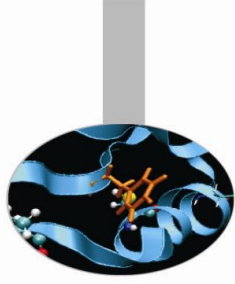


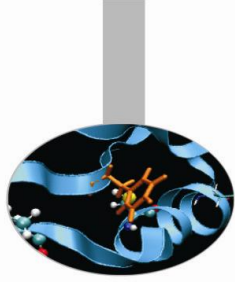
Winning strategies

Introduction



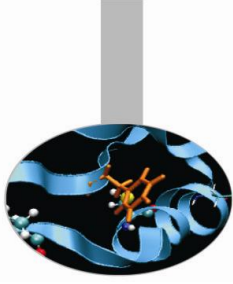
- Since most of serial applications may have several parallel solutions a methodological approach could be useful to evaluate the range of available strategies, to provide mechanisms for evaluating alternatives, and to reduce the cost of backtracking from bad choices.
  - The first step in developing parallel software is to understand the problem that you wish to solve in parallel looking at all the phases that can exploit parallelism.
  - If you are starting with an existing serial program, this necessitates understanding the existing code too.
  - Before spending time in an attempt to develop a parallel solution, determine whether or not the problem is one that can actually be parallelized.



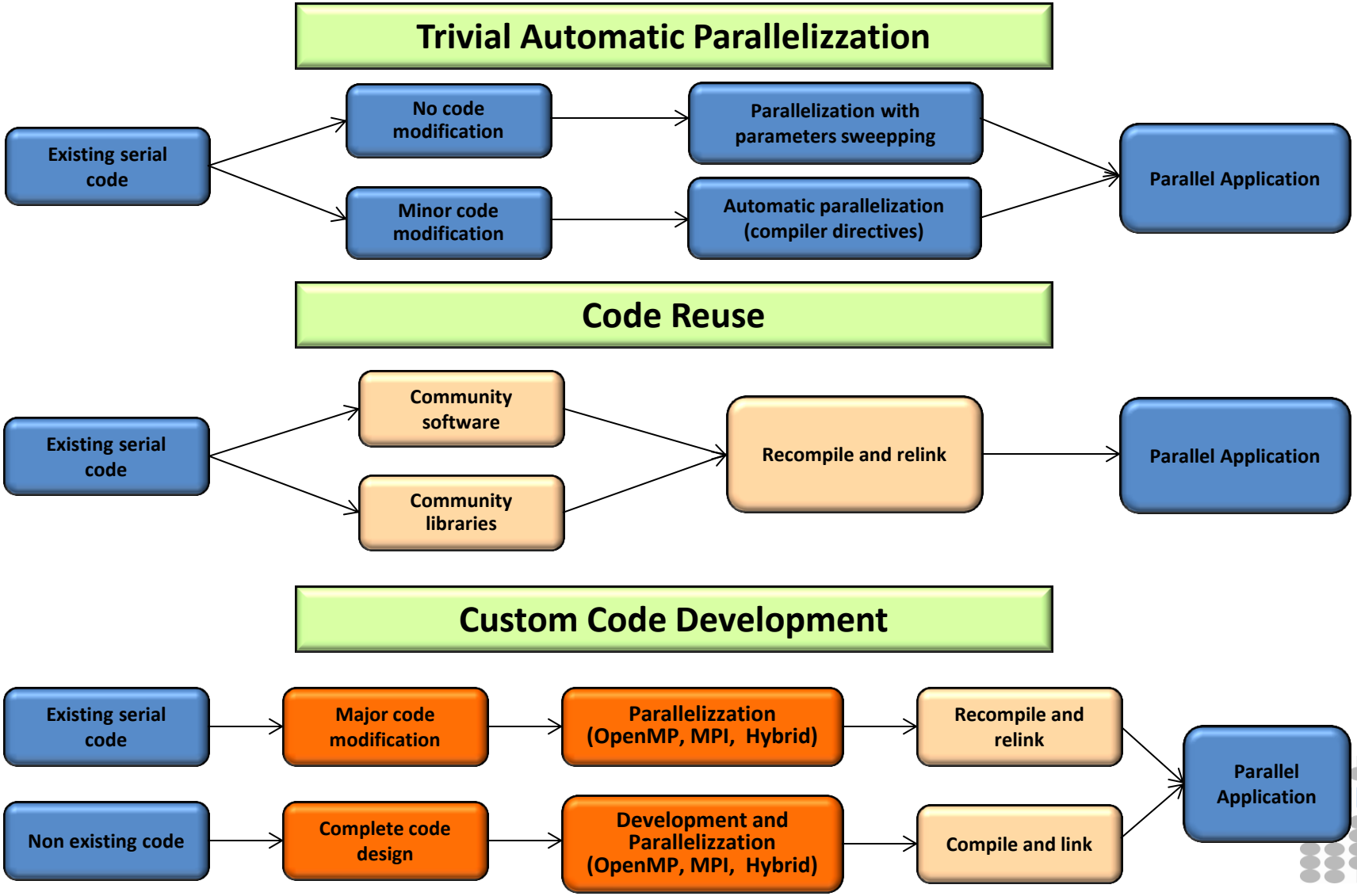
Taxonomy

Order is from less to most expensive in terms of time and parallelization complexity:

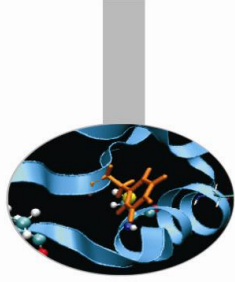
- Trivial parallelism (embarassing parallel)
- Automatic parallelization (compiler directives)
- community software (code reuse)
- community libraries (code reuse)
- custom code OpenMP (SMP exploitation)
- custom code MPI (perhaps with MPI I/O)
- custom code Hybrid (MPI & OpenMP)
- Accelerators: GPUs; languages: CUDA & OpenCL
- custom code Hybrid (MPI, OpenMP,CUDA,OpenCL)



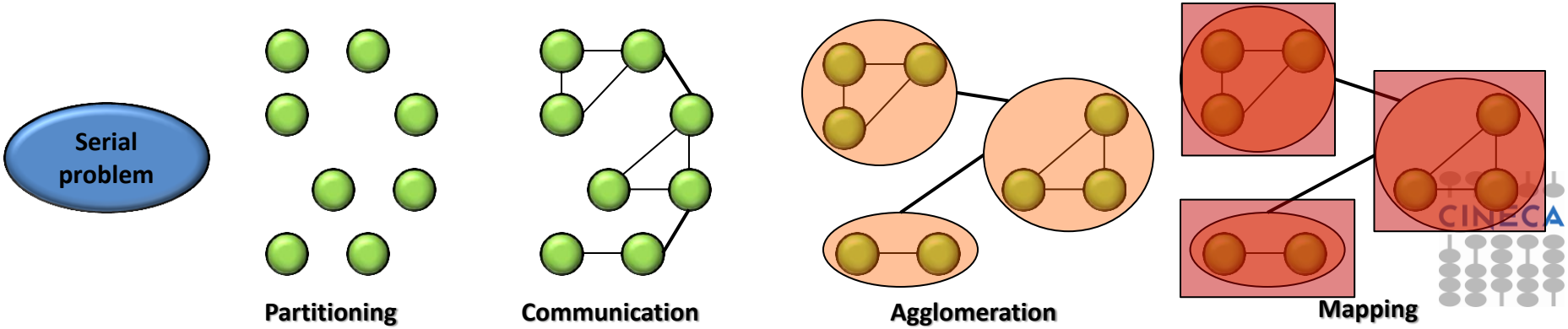
Taxonomy


Trivial Automatic Parallelizzation
Existing serial code
No code modification
Minor code modification
Parallelization with parameters sweepping
Automatic parallelization (compiler directives)
Parallel Application
Code Reuse
Existing serial code
Community software
Community libraries
Recompile and relink
Parallel Application
Custom Code Development
Existing serial code
Major code modification
Parallelization (OpenMP, MPI, Hybrid)
Recompile and relink
Non existing code
Complete code design
Development and Parallelization (OpenMP, MPI, Hybrid)
Compile and link
Parallel Application

Methodological approach



- Ian Foster strategy:
  - *Partitioning:*
    - The computation that is to be performed and the data operated on by this computation are decomposed into small tasks.
  - *Communication:*
    - The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.
  - *Agglomeration:*
    - The task and communication structures defined in the first two stages, if necessary, are combined into larger tasks to improve performance or to reduce development costs.
  - *Mapping:*
    - Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.



Partitioning

There are two basic ways to partition computational work among parallel tasks: ***domain decomposition*** and ***functional decomposition***.

Domain decomposition:

- In this type of partitioning, the data associated with a problem (input, output, intermediate values) is decomposed. Each parallel task then works on a portion of of the data.

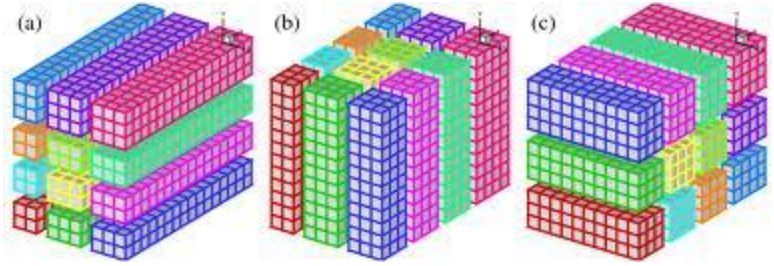


Fig. 1 Courtesy of Ning Li. Numerical *Algorithms Group (NAG)*

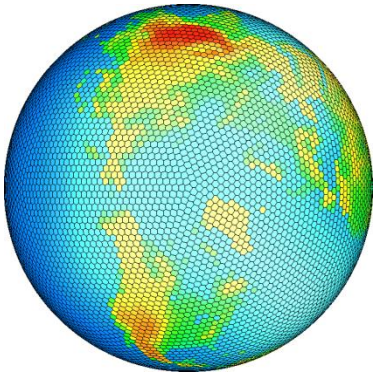


Fig. 2 Courtesy of Todd Ringler. *Los Alamos National Laboratory*

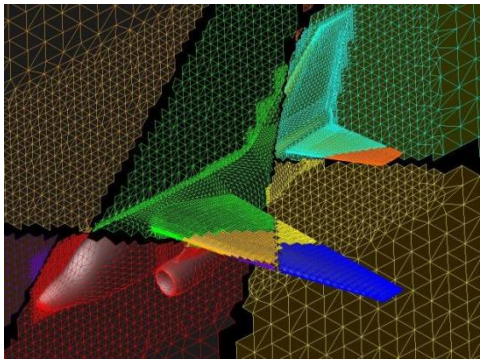
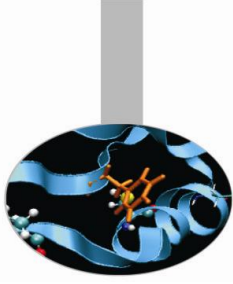


Fig. 3 Courtesy of Jaun Alonso. *Standford University*

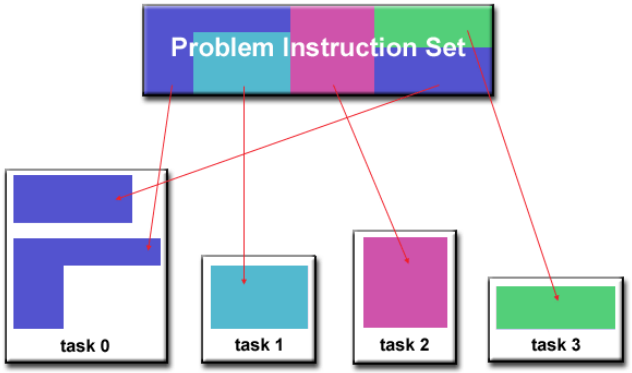
- Good rules of thumb are to focus first on the largest data structure or on the data structure that is accessed most frequently.
- Different phases of the computation may operate on different data structures or demand different decompositions for the same data structures. In this case, we treat each phase separately and then determine how the decompositions and parallel algorithms developed for each phase fit together.

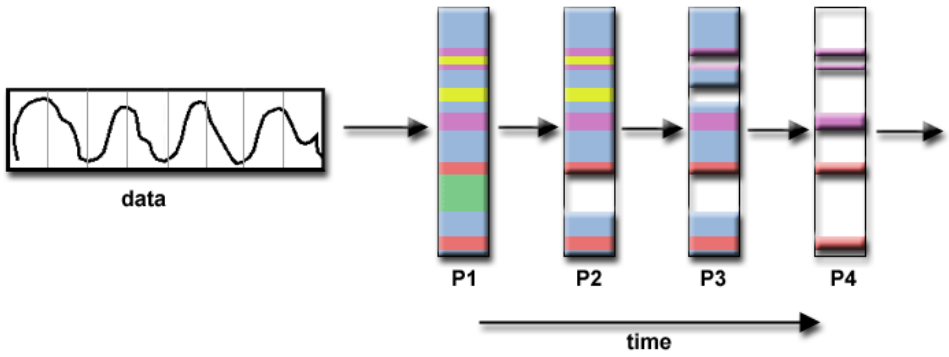


Partitioning

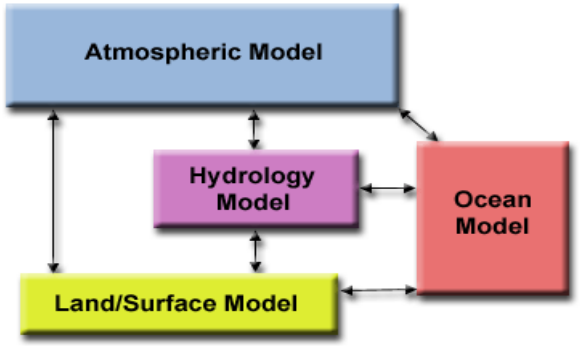
Functional decomposition

- In this approach, the initial focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- If we are successful in dividing this computation into disjoint tasks, we proceed to examine the data requirements of these tasks.





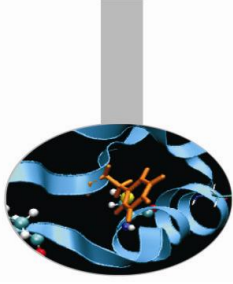
Signal processing



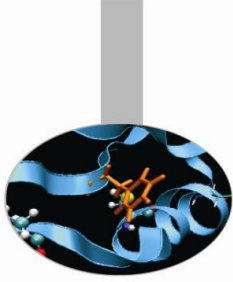
Atmospheric model

Fig. 4 Courtesy of Blaise Barney, *Lawrence Livermore National Laboratory*

Communication



- Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to communicate. These types of problems are often called ***embarrassingly parallel***
- Most of parallel applications are not quite so simple, and do require tasks to share data with each other. There are a number of important factors to consider when designing your program's inter-task communications:
  - Inter-task communication virtually always implies overhead.
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
  - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
  - Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.
  - Synchronous vs. asynchronous communications. Asyncronous communications are generally better because interleaving computation with communication could be a great benefit.



Agglomeration

- Domain and functional decomposition is a non trivial task which is exposed to the communication limit beetween processes.
- Communication cost among processes is one of the major limits to functional and domain decomposition.
- When communication exeeds computation time the parallel performance of the code is compromised and agglomeration of subdomains could be useful.

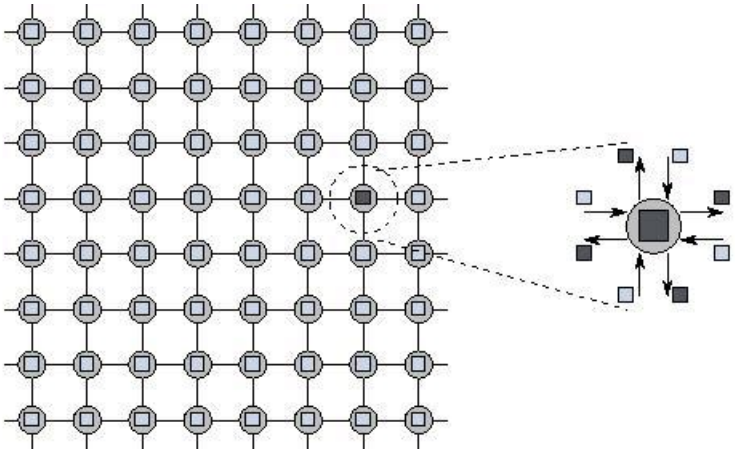


Fig. 5 Courtesy of Ian Foster. *Argonne National Laboratory*

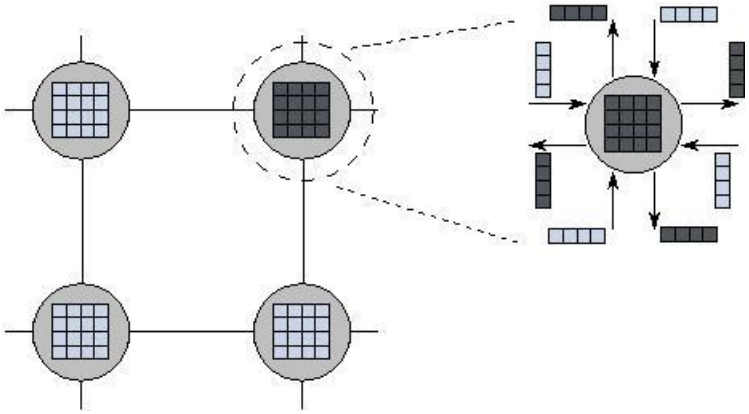
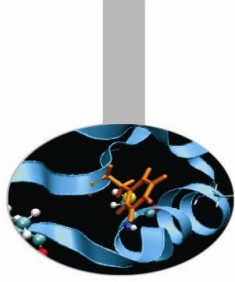


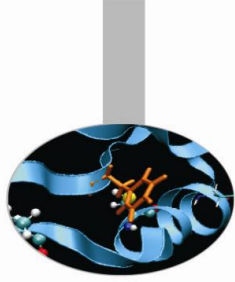
Fig. 6 Courtesy of Ian Foster. *Argonne National Laboratory*

In (Fig. 5), a computation on an 8x8 grid is partitioned into 64 tasks, each responsible for a single point, while in (Fig. 6) the same computation is partitioned on a 2x2 grid into 4 tasks, each responsible for 16 points.
In (Fig. 5), 256 communications are required, 4 per task; these transfer a total of 256 data values. In (Fig. 6), only 16 communications are required, and only 64 data values are transferred.



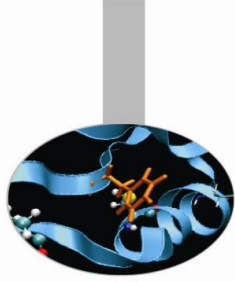
Mapping

- The goal of mapping techniques is normally to minimize total execution time. We use two strategies to achieve this goal:
  - We place tasks that are able to execute concurrently on *different* processors, so as to enhance concurrency.
  - We place tasks that communicate frequently on the *same* processor or node, so as to increase locality.
- Most common mapping techniques
  - Static mapping: many algorithms developed using domain decomposition techniques feature a fixed number of equal-sized tasks and structured local and global communication. In such cases, an efficient mapping is straightforward.
  - Dynamic mapping: in more complex domain decomposition-based algorithms with variable amounts of work per task and/or unstructured communication patterns, efficient agglomeration and mapping strategies may not be obvious. Hence, we may employ *load balancing* algorithms that seek to identify efficient agglomeration and mapping strategies, typically by using heuristic techniques. The time required to execute these algorithms must be weighed against the benefits of reduced execution time. The most complex problems are those in which either the number of tasks or the amount of computation or communication per task changes dynamically during program execution.



Methodological approach

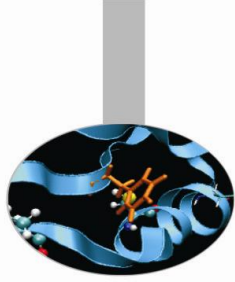
- Identify the program's ***hotspots***
  - Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
  - Profilers and performance analysis tools can help here
  - Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage
- Identify ***bottlenecks*** in the program
  - Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
  - May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas
- Identify **inhibitors** to parallelism
  - One common class of inhibitor is *data dependence*.
- Investigate **other algorithms** if possible
  - This may be the single most important consideration when designing a parallel application



Methodological approach

- Respect/be aware of **standards**
  - Programming: ANSI C, ISO C90/99, FORTRAN ISO 90 etc
  - Numerical: IEEE-754, IEEE 754-2008
  - System: POSIX compliance
- Respect/be aware of **scientific data formats**
  - HDF5 & BioHDF (this can help in Visualization, too)
  - NetCDF
  - GRIB, FITS, CERNLIB, XMDF et al
- Do Fault Tolerance and Verification & Validation
- Do checkpointing
  - Save the intermediate application states
- Documentation
  - Very important to ensure software quality

Bibliography



- Bibliography

- Designing and Building parallel programs, Ian Foster.
  An Online Publishing Project of Addison-Wesley Inc., Argonne National Laboratory, and the NSF Center for Research on Parallel Computation.
  http://www.mcs.anl.gov/~itf/dbpp/
- Introduction to Parallel Computing, *Blaise Barney, Lawrence Livermore National Laboratory*
  https://computing.llnl.gov/tutorials/parallel_comp/
- Spherical Geodesic Grids: A New Approach to Modeling the Climate, Todd Ringler, *Los Alamos National Laboratory*
  http://kiwi.atmos.colostate.edu/BUGS/geodesic/
- 2DECOMP&FFT – A highly scalable 2D decomposition library and FFT interface, Ning Li, *Numerical Algorithms Group (NAG)*
  http://www.2decomp.org/pdf/17B-CUG2010-paper-Ning_Li.pdf
- Computational Methods in Aircraft Design, Juan Alonso, *Standford University*
  http://adg.stanford.edu/aa241/design/compaero.html