Some aspects of parallel program design

R. Bader (LRZ)
G. Hager (RRZE)
Problem analysis

1. **Decompose into subproblems**
   - perhaps even hierarchy of subproblems
   - that can **simultaneously** execute
   - that still **efficiently** execute

2. **Identify dependencies between subproblems**
   - imply need for data exchange
   - identification of **synchronization** points
   - determine explicit communication between processors or ordering of tasks working on shared data
Example: climate simulation

- Prototypical for separate, but coupled systems
- Coarse grained parallelism
  - functional decomposition
    simulate various subsystems and/or observables
  - domain decomposition
    into equally-sized blocks
    each block contains observables on a grid
- Medium grained parallelism
  - further subdivide work
    if multiple CPUs available in a compute node
- Fine grained parallelism
  - instructions and data on a CPU
Load imbalance effects are hard to model in a general way.

Load imbalance may impede scalability much more strongly than communication or serial execution.

Two extreme cases:

A few “laggers” hurt a lot

A few “speeders” may not matter

Possible remedies for load imbalance:

- Dynamic load balancing
  - Scalable?
  - Overhead?
- Manual work distribution
  - General?
  - Inflexible?
- Choosing a different algorithm
  - Suitability?
Static load balancing (1)

- **Triangular MVM:**
  \[ y_i = \sum_{j=1}^{i} M_{ij} v_j \quad (i = 1, \ldots, N) \]
  - equi-distribution of iteration space (OpenMP default schedule) is imbalanced
  - some variants for rebalancing are possible (block-cyclic or dynamic scheduling)

- **optimal:**
  - reorganize iteration space to equi-partition „area“ (number of matrix entries)

- **Characteristic:**
  - precalculated user-defined scheduling
Static load balancing (2)

- **Irregular domain boundaries**
  - static load balancing using suitable discretization methods
  - e.g., triangulation (tetrahedral cells)

- **Calculate distribution of data to processors**
  - use a library
  - ParMETIS (freely available, but itself runs sequentially)
  - parallel partitioners and mesh generators needed for very large problems

- **Note:**
  - mesh generators are a big topic
Load balancing (task decomposition): Master-Worker

- **Conditions:**
  - unpredictable and variable workload per task – but otherwise independent tasks
  - cannot be mapped to loops (e.g., data stored in lists, trees etc.)
  - processors with differing capabilities may be available

**Trade-off:** granularity vs. distribution overhead

**Diagram:**
- Master (one)
  - initiate computation
  - set up problem
  - create bag of tasks
  - launch workers
  - sleep until work is done
  - collect results
  - terminate computation

- Workers (many)
  - initialize
  - compute result
  - done? (no)
  - compute result
  - done? (yes)
  - exit
References

- T. Mattson, B. Sanders, B. Massingill: Patterns for Parallel Programming
  - in-depth treatment of the abstract concepts
- A. Grama, A. Gupta, G. Karypis, V. Kumar: Introduction to Parallel Computing
  - details on many specific parallel algorithms
- ParMETIS: [http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview](http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview)
Parallel performance and scalability analysis

Georg Hager (RRZE)
Reinhold Bader (LRZ)
Understanding Parallelism: Sequential work

After 16 time steps: 4 cars
Understanding Parallelism: Parallel work

After 4 time steps: 4 cars
Understanding Parallelism: Specialization and Assembly-Line Work

After wind-up phase: 1 car per time step at lower cost!
Understanding Parallelism: Limits of Scalability

Unused resources due to load imbalance
Understanding parallelism:
Limits of scalability – shared resources + imbalance

- Unused resources due to resource bottleneck and imbalance!
- Waiting for shared resource
- Waiting for synchronization
Limits of Scalability: In search of simple models

Ideal world: all work is perfectly parallelizable

Reality: purely serial parts limit maximum speedup

Even worse: Communication processes hurt scalability even further
Limitations of Parallel Computing:
Calculating Speedup in a Simple Model ("strong scaling")

\[ T(1) = s + p = \text{serial compute time (=1)} \]

Parallelizable part: \( p = 1 - s \)

Purely serial part \( s \)

Parallel execution time:
\[ T(N) = s + \frac{p}{N} \]

General formula for speedup:
\[ S_p(N) = \frac{T(1)}{T(N)} = \frac{1}{s + \frac{1 - s}{N}} \]

Amdahl's Law for "strong scaling"
Benefit of parallelization is strongly limited

- "Side effect": limited scalability leads to inefficient use of resources
- Metric: Parallel Efficiency
  \( \text{(what percentage of the workers/processors is efficiently used)} \):

\[
\varepsilon_p(N) = \frac{S_p(N)}{N}
\]

Amdahl case:

\[
\varepsilon_p(N) = \frac{1}{s(N - 1) + 1}
\]
- Amdahl's law poses a serious limit on the use of parallel resources!
  - Example: 1024 workers, k=0

Serial fraction = 1 %

max. speedup=100
Limits of Scalability: Strong Scaling with a Simple Communication Model

\[ T(1) = s + p = \text{serial compute time} \]

parallelizable part: \( p = 1 - s \)

\[ T(N) = s + \frac{p}{N} + Nk \]

fraction \( k \) for communication between each two workers (only present in parallel case)

General formula for speedup (worst case):

\[ S_p(N, k) = \frac{T(1)}{T(N, k)} = \frac{1}{s + \frac{1-s}{N} + Nk} \]

\( k=0 \Rightarrow \text{Amdahl's Law} \)
Limits of Scalability: Amdahl's Law

- **Large N limits**
  - at $k=0$, Amdahl's Law predicts
    $$\lim_{N \to \infty} S_p(N, 0) = \frac{1}{s}$$
    (independent of $N$)
  - at $k \neq 0$, our simplified model of communication overhead yields a behavior of
    $$S_p(N, k) \xrightarrow{N \gg 1} \frac{1}{Nk}$$
Understanding Parallelism: Amdahl’s Law and the impact of communication

Parallelism efficiency $S(N)$ as a function of the number of CPUs $N$.

- $s=0.01$
- $s=0.1$
- $s=0.1, k=0.05$
Communication is not necessarily purely serial

- **Non-blocking** crossbar networks can transfer many messages concurrently – factor $Nk$ in denominator becomes $k$ (technical measure)
- Sometimes, **communication can be overlapped** with useful work (implementation, algorithm):
Limits of Scalability: How to circumvent the limitations

- The more workers there are, the smaller each share of work gets (optimization is easier; large aggregate cache size)
  - "superlinear speedups" possible if communication is negligible

- The larger the problem, the larger the parallelizable fraction of execution time may become
  - i.e., $s$ gets smaller and the Amdahl Limit is shifted
  - See below for “Gustafson’s Law”

- Communication overhead may scale with a smaller power than problem size
  - See below for refined performance models
• Increasing problem size often helps to enlarge the parallel fraction $p$
  • Often $p$ scales with problem size while $s$ stays constant
  • Fraction of $s$ relative to overall runtime decreases

Scalability in terms of parallel speedup and parallel efficiency improves when scaling the problem size!
When scaling a problem to more workers, the amount of work will often be scaled as well

- Let $s$ and $p$ be the serial and parallel fractions so that $s + p = 1$
- Perfect situation: runtime stays constant while $N$ increases
- “Performance Speedup” =

\[
Ps(N) = \frac{s + pN}{s + p} = s + (1 - s)N
\]

Gustafson's Law
("weak scaling")

- Linear in $N$ – but closely observe the meaning of the word "work"!

Also works for strong scaling!
Serial fraction $s$ may depend on

- **Program / algorithm**
  - Non-parallelizable part, e.g. recursive data setup
  - Non-parallelizable IO, e.g. reading input data
  - Communication structure
  - Load balancing (assumed so far: perfect balanced)
  - …

- **Computer hardware**
  - Processor: Cache effects & memory bandwidth effects
  - Parallel Library; Network capabilities; Parallel IO
  - …

**Determine $s$ "experimentally":**
Measure speedup and fit data to Amdahl’s law – but that could be more complicated than it seems…
Scalability data on modern multi-core systems

An example

Scaling across nodes

1 → 2 sockets on node

1 → 2 cores on socket

Affinity matters!
Scalability data on modern multi-core systems

The scaling baseline

Scalability presentations should be grouped according to the largest unit that the scaling is based on (the “scaling baseline”)

Scalability

Amdahl model w/ communication: Fit

\[ S_p(N, k) = \frac{1}{S + \frac{1-s}{N} + Nk} \]

to inter-node scalability numbers (\(N = \# \text{ nodes}, >1\))
Application to accelerator devices

- GPUs, Cell SPEs, FPGAs, Clearspeed,…
- Assume overall (serial, un-accelerated) runtime to be \( T_s = s + p = 1 \)
- Assume \( p \) can be accelerated and run on special hardware \( \alpha \) times faster. We neglect any communication cost.
- To get a speedup of \( r\alpha \), how small must \( s \) be? Solve for \( s \):

\[
r\alpha = \frac{1}{s + \frac{1-s}{\alpha}} \quad \Rightarrow \quad s = \frac{r^{-1} - 1}{\alpha - 1}
\]

- At \( \alpha = 10 \) and \( r = 0.9 \) (for an overall speedup of 9), we get \( s \approx 0.012 \), i.e. you must accelerate almost 99% of the serial runtime!

- Limited memory on accelerators may limit the achievable speedup
Refined performance models
A more general formulation

- Let \( s+p=1 \) be the serial work and runtime; we assume there is no communication overhead in this case.
- In the parallel case, the parallel problem size is scaled with \( N^\alpha \), so that parallel runtime is

\[
T_p = s + pN^{\alpha-1} + c_\alpha(N), \quad 0 \leq \alpha \leq 1
\]

- The parallel speedup is then

\[
S(N) = \frac{s + (1 - s)N^\alpha}{s + (1 - s)N^{\alpha-1} + c_\alpha(N)}
\]

- \( \alpha=0 \): Strong scaling (\( \rightarrow \) Amdahl’s Law)
- \( \alpha=1 \): Weak scaling (\( \rightarrow \) Gustafson’s Law)
**Overhead**

- **Some typical cases for** $c_\alpha(N)$
  
  - Bus network, every worker sends one message:
    
    $$c_\alpha(N) = (\kappa + \lambda)N$$
  
  - Fully non-blocking network, every worker sends one message:
    
    $$c_\alpha(N) = \kappa + \lambda$$
  
  - Overhead for $N$-way mutual sync (“barrier”):
    
    $$c_\alpha(N) = \lambda \log N$$
  
  - Fully non-blocking network, $d$-dimensional stencil code (halo exchange):
    
    $$c_\alpha(N) = \kappa \left[ \left( \frac{N^\alpha}{N} \right)^{1/d} \right]^{d-1} + \lambda = \kappa N^{-\beta} + \lambda, \text{ with } \beta = \frac{(\alpha - 1)(d - 1)}{d}$$

$\kappa$ - Bandwidth-determined transfer time

$\lambda$ - Latency
Choices for 3D domain decomposition: Communication volume per worker for $N$ workers

"Slabs"
\[ c_{1d}(L,N) = L \cdot L \cdot w \cdot 2 \]
\[ = 2wL^2 \]
Constant cost with growing $N \rightarrow$ hazardous!

"Poles"
\[ c_{2d}(L,N) = L \cdot \frac{L}{\sqrt{N}} \cdot w \cdot (2 + 2) \]
\[ = 4wL^2N^{-1/2} \]
Decreasing cost with growing $N \rightarrow$ better!

"Cubes"
\[ c_{3d}(L,N) = \frac{L}{\sqrt[3]{N}} \cdot \frac{L}{\sqrt[3]{N}} \cdot w \cdot (2 + 2 + 2) \]
\[ = 6wL^2N^{-2/3} \]
Faster decreasing cost with growing $N \rightarrow$ best!

Whatever you do, communication will always kill you at strong scaling!
“Slow Computing”

Why slower codes/computers scale better

- What happens if we slow down the execution of the code by some factor $\mu$?

\[ S(N) = \frac{(s + (1 - s)N^\alpha) \cdot \mu}{(s + (1 - s)N^{\alpha-1}) \cdot \mu + c_\alpha(N)} \]

\[ \Rightarrow S(N) = \frac{s + (1 - s)N^{\alpha}}{s + (1 - s)N^{\alpha-1} + c_\alpha(N)/\mu} \]

- The communication overhead is damped by a factor of $\mu$ in the speedup
  - “Slow computers” (i.e., with slow CPUs) scale better
  - “Slow code” also scales better! \( \Rightarrow \) see pizza talk tonight!
Summary on parallel scalability

- **Scalability is not the only goal**
  - Node-level performance comes first
  - Check your performance models!
  - Scaling slow code is easy

- **Be aware of sequential/redundant work (Amdahl’s Law!)**
- **Load imbalance should be fixed first**
- **Be aware of communication vs. computation ratio**
  vs. number of workers and problem size
  - Slow computers and programs scale better!

- **Separation of bottlenecks** is paramount when analyzing performance data
  - Node-level bottlenecks are typically different from inter-node bottlenecks