Parallel Programming

Timings, pt.3

Prof. Paolo Bientinesi

HPAC, RWTH Aachen
pauldj@aices.rwth-aachen.de

WS17/18

Time

- **Wall time** or “wall-clock time” :
  real time between the beginning and the end of a computation

- $T_p(n)$:= Wall time to solve a problem of size n using p processes

  $T_p(n) = t_1 - t_0$
  t_0 : time when the first process starts its execution,
  t_1 : time when the last process completes its execution

- **CPU-time** or “core time” :
  cumulative time spent by all processes in a computation

Code

- `time0.c`
  Cholesky factorization.
  No timings. Only correctness.

- `time1.c`
  Timings through `clock()`.
  Multithreading (via LAPACK/BLAS). CPU-time.

- `time2a.c`
  Cycle accurate timer.
  Cycles, frequency. Wall time vs. CPU-time.

- `time2b.c`
  Performance (# ops/sec), efficiency.

Performance

- **Performance**: Number of floating point operations per second performed while solving a given problem.
- **Theoretical Peak Performance** (TPP): In ideal conditions, the highest number of floating point operations that a processor can perform in one second.
- **Peak Performance** “Practical peak performance”: The performance attained by highly tuned matrix-matrix multiplication kernels (DGEMM). For instance, MKL and OpenBLAS.
- **Efficiency**: The ratio between the performance attained while solving a given problem and the TPP (or the PPP).

Code

- `time3.c`
  GEMM as a triple loop.
  Terrible performance and efficiency. No parallelism.

Speedup

- **Speedup**: $S_p(n) := \dfrac{T_1(n)}{T_p(n)}$ Typically: $0 \leq S_p(n) \leq p$

  If $S_p(n) > p$: "superlinear speedup" $\leftarrow$ rare

- **What is $T_1(n)$?** Time of the best sequential code.
  NOT the time for the parallel code run with $p = 1$!!

  Note: The sequential code could possibly implement a different algorithm than the parallel one.

  Example: Eigenvalues of symmetric tridiagonal matrix
  - Alg.1: `dqds` cost: $O(n^2)$, BUT inherently sequential
  - Alg.2: `BX` cost: $O(n^3)$, BUT perfectly parallelizable

Speedup, Efficiency

- **What if p is large?** Then probably $T_1(n)$ is not obtainable, either because n is too large a problem to be solved sequentially, or because it would take too long to complete.

  In this case, $S_p(n) := \dfrac{T_{p_0}(n)}{T_p(n)}$, with $0 \leq S_p(n) \leq \dfrac{p}{p_0}$.

  $T_{p_0}(n)$ is used as a reference, possibly from a different code.

  Note: excellent speedup does not imply excellent performance.

- **Parallel efficiency**:

  $E_p(n) := \dfrac{S_p(n)}{p} \quad \left(\text{or } \; E_p(n) := \dfrac{S_p(n)}{p/p_0} \right) \qquad 0 \leq E_p(n) \leq 1$

  Note: excellent efficiency does not imply excellent performance.

Scalability

- **Strong Scalability**: Behaviour of $T_k(n)$, as k increases.
  Fixed problem size, increasing number of processes.

  Example: $\bar{n} = 24k$; $p = 2^i$, with $i \in [10, \ldots, 14]$

  $\Rightarrow$ $T_{1024}(\bar{n})$, $T_{2048}(\bar{n})$, $T_{4096}(\bar{n})$, $T_{8192}(\bar{n})$, $T_{16384}(\bar{n})$

- **Weak Scalability**: Behaviour of $T_k(n)$, as n and k increase to keep the memory usage per process constant.
  Fixed memory load per process, increasing problem size and number of processes.

  Example: Algorithm $\mathcal{A}$; input: $n \in \mathbf{N}$
  Time complexity: $O(n^3)$ Space complexity: $O(n^2)$
  Reference: $\bar{n} = 100$, $\bar{p} = 16$; $p = 2^i$, with $i \in [4, 6, 8, 10]$

  $\Rightarrow$ $T_{16}(100)$, $T_{64}(200)$, $T_{256}(400)$, $T_{1024}(800)$

Weak Scalability

- **Example #1**: Algorithm $\mathcal{B}(n)$
  Time($\mathcal{B}(n)$): $O(n^2)$, Space($\mathcal{B}(n)$): $3n$, Reference: $T_p(n) = t_0$.

  - $T_{??}(2n)$ (from problem size to number of processors)
  - Space(n) = $3n$ $\Rightarrow$ Mem/proc: $3n/p$ == *const*
  - Space($2n$) = $6n$ $\Rightarrow$ $6n/??$ == *const* $\Rightarrow$ $?? = 2p$
  - $T_{2p}(2n) = t_1$ $t_0 > / = / < t_1$?
  - assuming perfect scalability:
    $t_1 = T_{2p}(2n) \approx 4\,T_{2p}(n) \approx 4\,T_p(n)/2 = 2t_0$

  - $T_{2p}(??)$ (from number of processors to problem size)
  - Space(n) = $3n$ $\Rightarrow$ Mem/proc: $3n/p$ == *const*
  - Space(??) = $3\,??$ $\Rightarrow$ Mem/proc: $3\,??/2p$ == *const* $\Rightarrow$ $?? = 2n$

Weak Scalability

- **Example #2**: Algorithm $\mathcal{B}(n)$
  Time($\mathcal{B}(n)$): $O(n^2)$, Space($\mathcal{B}(n)$): n^2, Reference: $T_p(n) = t_0$.

  - $T_{??}(2n)$
  - Space(n) = n^2 $\Rightarrow$ Mem/proc: n^2/p == *const*
  - Space($2n$) = $4n^2$ $\Rightarrow$ $4n^2/??$ == *const* $\Rightarrow$ $?? = 4p$
  - $T_{4p}(2n) = t_1$ $t_0 > / = / < t_1$?
  - assuming perfect scalability:
    $t_1 = T_{4p}(2n) \approx 4\,T_{4p}(n) \approx 4\,T_p(n)/4 = t_0$

  - $T_{2p}(??)$
  - Space(n) = n^2 $\Rightarrow$ Mem/proc: n^2/p == *const*
  - Space($??$) = $??^2$ $\Rightarrow$ Mem/proc: $??^2/2p$ == *const* $\Rightarrow$
    $??^2 == 2pn^2/p$ $\Rightarrow$ $?? == n\sqrt{2}$

Code

- `time4.c`
  Timings breakdown: Malloc, init, compute, test.
  Scalability. Serial vs. parallel code. Amdahls law.

Amdahl's law

Maximum possible speedup when only a portion of the code scales.

- T_{seq}: strictly sequential portion of the algorithm (in secs)
- T_{par}: parallel portion of the algorithm (in secs)
- $T_1(n) == T_{seq} + T_{par}$
- β: fraction of the algorithm that is strictly sequential
- $\beta == \dfrac{T_{seq}}{T_{seq} + T_{par}}$
- $T_p(n) == \beta T_1(n) + (1 - \beta)\, T_1(n)/p == T_1(n) \left(\beta + \frac{(1-\beta)}{p} \right)$
- $S_p(n) == \dfrac{T_1(n)}{T_1(n)\left(\beta + \frac{(1-\beta)}{p}\right)} == \dfrac{1}{\beta + (1-\beta)/p}$ $\qquad \lim_{p \to \infty} S_p(n) = 1/\beta$