Lecture 5:

OpenMP and Performance Evaluation

Erich Strohmaier
UT
CS594
Feb 7, 2001

HPF and OpenMP

- Two standards for parallel programming without explicit message passing.
- HPF targets data parallel applications for distributed memory MIMD systems.
- OpenMP targets (scalable) shared memory Multiprocessor.

HPF - High Performance Fortran

- Standard for data parallel applications for distributed memory MIMD systems.
- Idea: "Make it easy for the programmer and let the compiler do the hard work".
- Influenced by: Fortran D, Vienna Fortran, (CM Fortran, MasPar Fortran, C*, …)
- Failed to gain broad acceptance.

OpenMP

- Portable, Shared Memory MultiProcessing API.
  - Fortran 77 & Fortran 90 and C & C++.
  - Multi-vendor support, for both UNIX and NT.
- Standardizes fine grained (loop) parallelism.
- Also supports coarse grained algorithms.
- Based on compiler directives and runtime library calls.

OpenMP

- Specifies:
  - Work-sharing constructs.
  - Data environment constructs.
  - Synchronization constructs.
  - Library Routines and Environment Variables.

OpenMP: Work Sharing

- PARALLEL
- DO
- SECTION
- SINGLE

OpenMP: Parallel Region

- Fundamental parallel construct in OpenMP
  - !$OMP PARALLEL [clause[[,] clause] . . .]
    statement-block
    !$OMP END PARALLEL
  - Team of threads gets created.
  - *Statement-block* gets executed by multiple threads in parallel.
  - Implied barrier at 'END PARALLEL'.
  - Is needed around 'DO' and 'SECTIONS'.

OpenMP: Parallel Region

- Clause can be:
  - PRIVATE (var-list)
  - SHARED (var-list)
  - DEFAULT (PRIVATE | SHARED | NONE)
  - FIRSTPRIVATE (var-list)
  - REDUCTION ({operator|intrinsic} : var- list)
  - IF (scalar_logical_expression)
  - COPYIN (list)

OpenMP: DO

- Distributes loop iterations across the available threads in a parallel region.
  - !$OMP DO [clauses]
    Fortran do loop
    [!$OMP END DO [NOWAIT]]
  - Implicit barrier at end (unless NOWAIT).
  - Shortcut for single loop in region:
    - !$OMP PARALLEL DO [clauses] ...

OpenMP: DO

- Clauses:
  - PRIVATE (var-list)
    FIRSTPRIVATE (var- list)
    LASTPRIVATE (var- list)
    REDUCTION ({operator|intrinsic} : var- list)
    SCHEDULE (type[,chunk])
    ORDERED

OpenMP: SECTIONS

- Distributes sections of code among threads.
  - !$OMP SECTIONS [clauses]
    [!$OMP SECTION]
    block
    [!$OMP SECTION]
    block
    . . .
    !$OMP END SECTIONS [NOWAIT]
  - "Shortcut possible and barrier implied"

OpenMP: SECTIONS

- Clauses:
  - PRIVATE (var-list)
    FIRSTPRIVATE (var- list)
    LASTPRIVATE (var- list)
    REDUCTION ({operator|intrinsic} : var- list)

OpenMP: SINGLE

- Enclosed code is executed by only one thread in the team.
  - !$OMP SINGLE [clauses]
    block
    !$OMP END SINGLE [NOWAIT]
  - Other thread wait at END (unless NOWAIT)
  - Clauses:
    - PRIVATE (var-list)
      FIRSTPRIVATE (var- list)

OpenMP: Data Environment

- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- SHARED
- DEFAULT
- REDUCTION
- SHEDULE
- THREADPRIVATE
- COPYIN

OpenMP: PRIVATE

- PRIVATE (var-list)
  - Declares elements of list as private to thread.
  - Each thread uses its own copy.
  - Copies are undefined at entry to the parallel construct and after exit!
  - !$OMP PARALLEL DO PRIVATE(i) SHARED(xnew, xold, dx)
    do i = 2, n
    xnew(i) = (xold(i) - xold(i-1)) / dx
    enddo
    !$OMP END PARALLEL DO

OpenMP: PRIVATE

- FIRSTPRIVATE (var-list)
  - Private variables with initialisation:
    - !$OMP PARALLEL DO SHARED(c, n) PRIVATE (i, j)
      !$OMP& FIRSTPRIVATE(a)
- LASTPRIVATE (var-list)
  - Private variables which are defined after parallel execution.
    - Value determined by sequentially last iteration.

OpenMP: REDUCTION

- REDUCTION ({ operator | intrinsic } : list)
  - Operator: +, *, -, .AND., .OR., .EQV., .NEQV.
  - Intrinsic: MAX, MIN, IAND, IOR, IEOR
  - !$OMP PARALLEL
    !$OMP DO SHARED (a, n) PRIVATE (i) REDUCTION (max:maxa)
    do i = 1, n
    maxa = max (maxa , a)
    enddo
    !$OMP END PARALLEL

OpenMP: Synchronization

- MASTER
- CRITICAL
- BARRIER
- ATOMIC
- FLUSH
- ORDERED

OpenMP: Synchronization

- !$OMP MASTER and END MASTER
  - Code enclosed is executed by the master thread.
  - No implied barriers around it!
- !$ OMP CRITICAL and END CRITICAL
  - Restricts access to the enclosed code (critical section) to one thread at a time.

OpenMP: Synchronization

- !$ OMP BARRIER
  - Synchronize all the threads by causing them to wait.
  - Must be encountered by all threads in a team or by none at all.
- !$ OMP ATOMIC
  - Specified memory location is updated atomically.
  - Applies only to the immediately following statement.
  - Restrictions apply

OpenMP: Synchronization

- !$ OMP FLUSH
  - Tthread visible variables, (e.g. common blocks, pointer dereferences), are written back to the memory.
  - (Only if you write your own synchronization types)
- !$OMP ORDERED and END ORDERED
  - Only inside of 'DO'.
  - Critical section with implied order in which the thread can enter (sequential order).

HPF and OpenMP links

- HPF:
  - www.crpc.rice.edu/HPFF/home.html
  - www.vcpc.univie.ac.at/activities/tutorials/HPF
- OpenMP:
  - www.openmp.org
  - www.epcc.ed.ac.uk/epcc-tec/documents/techwatch-openmp/report-2.html

Performance Modeling

- The three traditional performance evaluation techniques are:
  - Analytic Modeling
  - Simulation
  - Measurement

Performance Modeling: Analytic

- Pure Analytic:
  - Based on theoretical operation count of algorithm and it's implementation.
  - Duration of individual operations has to be known (estimated).
  - Establish algebraic timing formula.
  - Can be done without the computer system in question.
  - Accuracy in many cases low.

Performance Modeling: Analytic

- Queueing Theory
  - Views system as collection of resources.
  - Jobs wait in queues to get access to resources.
  - Predicts total time jobs remain in system.

Performance Modeling: Simulation

- Emulation - uses hardware simulator.
- Monte Carlo - simulates equilibrium states, no time dependencies.
- Trace-Driven - 'time ordered record of events'.
- Discrete Event - discrete internal states of systems only.

Measurement based

- Measurement based analytic modeling:
  - Algebraic timing formula based on theoretical time dependencies.
  - Free parameters in model.
  - Needs measurements to calibrate parameter.
  - Often used.

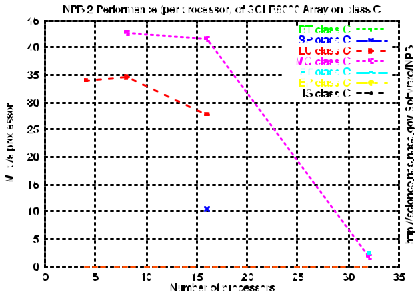
Preparation of Data

- Inspection of data is necessary to ensure the quality of any statistical analysis.
- Visual inspection of data to check:
  - Quality of data.
  - Relation between independent and dependent variables.
  - Scope of assumed relation, modes of operation of system, ...

Preparation of Data

- Search for outliers:
  - They have to be excluded from the analysis
- Check if transformation of measured data is necessary!

Preparation of Data: Outliers



Preparation of Data: Transformation

- The term transformation is used when some function of the measured variable is analyzed eg:
  - $\sqrt{y} = b_0 + b_1 x$
  - $\log(y)$

Preparation of Data: Transformation

- Transformations maybe necessary for the following reasons:
  - Physical relations
  - Range of measured values too large (several orders of magnitude)!
  - Distribution of residuals is not homogeneous
- Caveat: The inverse function does not help!

Statistics: Mean Values

- (Arithmetic) mean: $\bar{y} = \frac{\sum y}{n}$
  - strongly influenced by large values
- Harmonic mean: $\bar{y} = \frac{n}{\sum \frac{1}{y}}$
  - strongly influenced by small values
- Geometric mean: $\bar{y} = \left(\prod y \right)^{\frac{1}{n}}$
  - has no physical meaning - often used

Statistics: Variability

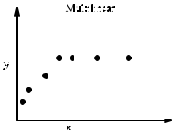
- Variance: $s^2 = \frac{1}{n-1} \sum (y_i - \bar{y})^2$
- Standard Deviation: s
- Skewness, Kurtosis: $sk = \frac{1}{ns^3} \sum (y_i - \bar{y})^3$

Linear Regression

- Linear (and non-linear) regression is *the* essential technique for all measurement based performance modeling:
  - Linear Regression - Basic Example
  - Parameter Estimates
  - Allocation of Variation
  - Confidence Intervals of Parameters
  - Confidence Intervals of Predictions
  - Testing Assumptions about Regressions

Linear Regression: Parameter Estimates

- Always check visually if assumptions are valid?



- Best fitting line determined by minimal Squared Error:
  - Estimated value: $\hat{y}_i = b_0 + b_1 x_i$
  - Error of estimate: $e_i = y_i - \hat{y}_i$

Linear Regression: Parameter Estimates

- Parameters are computed by:
  - Slope: $b_1 = \frac{\sum xy - n\bar{x}\bar{y}}{\sum x^2 - n\bar{x}^2}$
  - Intersection: $b_0 = \bar{y} - b_1\bar{x}$

Common Simple Timing Models

- Linear timing models
- Hockney Parameter
- Amdahl's Law
- Gustafson's Law
- Pipelines
- Communication
- others

Simple Models: Linear Timing Models

- Many timing models express the execution time of a function as a linear combination of a start-up and an execution phase. The time spent in the execution phase depends typically on resource parameters (number of processors, multiplicity of pipelines, ...) and problem size parameters (Matrix sizes, vector length, message length, ...):
  - $t_m = t_0 + t_c(r,m)$
  - t_0 is often independent of resources (r) or problem size (m)
  - t_c depends often linear or inverse on resources or problem size

Simple Models: Hockney Parameter

Originally developed for vector units; can be applied more generally.

- Assumption: constant time per operation in the execution phase
- $t_m = t_0 + t_c(m)$
- t_0 : Startup
- t_c : Speed in execution phase

Simple Models: Hockney Parameter

Usually formulated in the following way.

- $t_m = t_0 + t_c(m)$
- $r_\infty = 1/t_c$: Speed for large problem size
- $m_{1/2} = t_0 / t_c$: half-performance problem size
- m can be any metric for the problem size

$$Perf = \frac{m}{t_m} = \frac{r_\infty}{1 + \frac{m_{1/2}}{m}}$$

Simple Models: Amdahl's Law

- Originally developed for single processor performance on mainframes; can be applied more generally; nowadays usually formulated for parallel systems.
- Assumption: time in the execution phase inverse to available resources
- $t_p = t_0 + t_c \frac{1}{p}$
- Interpretation: We have only two types of work: ideal parallel and totally serial; their relative ratio is fixed (eg. the problem size itself is fixed)

Simple Models: Amdahl's Law

- Two formulations are common: $t_p = t_0 + t_c \frac{1}{p}$
- $r_\infty = \frac{1}{t_0}$: Speed for large problem size (t_0 NOT t_c!)
- $p_{\frac{1}{2}} = \frac{t_c}{t_0}$: half-performance system size
- p could be any other resource metric

$$Perf = \frac{1}{t_p} = \frac{r_\infty}{1 + \frac{p_{\frac{1}{2}}}{p}}$$

Simple Models: Amdahl's Law

- Other formulation sometimes useful: $t_p = t_0 + t_c \frac{1}{p}$
- $r_1 = \frac{1}{t_0 + t_c}$: Speed on one processor
- $\alpha_1 = \frac{t_c}{t_0 + t_c}$: Parallelization ratio

$$Perf = \frac{1}{t_p} = \frac{r_1}{(1 - \alpha_1) + \frac{\alpha_1}{p}}$$

$$\lim_{p \to \infty} Perf = \frac{1}{1 - \alpha_1}$$

Simple Models: Gustafson's Law

- Was a reply to the critics using Amdahl's Law arguing against parallel systems.
- Idea: We have only two types of work: ideal parallel and totally serial; their relative ratio is not fixed but changes with the problem size.
- Assumption: parallel work scales with the problem size while the serial works remains constant!

$$t_{m,p} = t_0 + t_c \frac{m}{p}$$

Simple Models: Gustafson's Law

- We now scale the problem size with the number of processors $m=p$ (memory?)
- $t_m = t_0 + t_c = const$

$$Perf = \frac{t_0 + t_{c\,m}}{t_{m,p}} = r_1 \cdot (1 - \alpha_1 + \alpha_1 * p)$$

- For large processor numbers we always get linear Speedup!

Simple Models: Pipelines

- We assume optimal operation:
- Start-up phase: t_0 fixed
- Execution phase: one result per cycle
- $t_n = t_0 + t_c\, n$
- ? Hockney

Simple Models: Parallel Pipelines

- Now for parallel (vector) processing:
- Start-up phase does not parallelize: t_0 fixed
- Execution phase parallelizes: one result per cycle per processor
- $t_{n,p} = t_0 + t_c * n/p$
- p fixed: 'Hockney' but r_∞ and $n_{1/2}$ are inverse proportional to p!
- n fixed: Amdahl with respect to p
- n/p fixed: Gustafson

Simple Models: Communication

- Critical question is how the amount of communicated data depend on the processor number?
- $t_{n,p} = t_0 + t_c * n/p$
- n now represents the total amount of data communicated!
- Discussion just like parallel pipelines
- ....
- More refined model: Log(P)

Simple Models: Parallel Work

- This type of work is defined to be ideally parallelizable on all processors.
- $t_p ? \frac{s_p(N)}{r_p}\frac{1}{p}$ with $u_p ? \frac{1}{p}$
- $s_p(N)$: Amount of work depending on problem size and processor number
- $1/p$: Speed for this type of work.
- u_p: Function characterizing the dependency of the performance from the processor number.

Simple Models: Sequential Work

- Here the execution time is independent of the number of processors used

$$t_s ? \frac{s_s(N)}{r_s} \quad \text{with} \quad u_s ? 1$$

Simple Models: Global Overhead

- This is a general overhead and thus constant. It differs from sequential work as no useful work is done. Thus the unit of work is different.

$$t_O ? const \quad \text{with} \quad u_O ? 1$$

Simple Models: Reduction Operation

- Here we assume that $N ? p$. Then the first part of the computation can be done in parallel. The last p operations are done in a binary tree-like operation.

$$t_r ? \frac{s_r(N)}{r_r}\frac{?}{?}\frac{N ? p}{p} ? \frac{\log(p)}{p}\frac{?}{?} \quad \text{with} \quad u_r ? \frac{\log(p)}{p}$$

Simple Models: Limited parallelism

- If the maximum amount of parallelism in the calculation can be smaller than the number of processors, we have to distinguish two possible cases. We formulate this by using the step function ? (x), which is 0 if x? 0 and 1 otherwise.

$$t_{pl} ? \frac{s_{pl}(N)}{r_{pl}}\frac{1}{p_{crit} ? ?p_{crit} ? p?* ? ?p_{crit} ? p?}$$

Simple Models: Broadcast

- If we assume that the broadcast is implemented in a binary tree communication pattern, the execution time goes up with *log(p)*

$$t_b ? \frac{s_b(N)}{r_b} \log(p) \quad \text{with} \quad u_r ? \log(p)$$

Point to Point Communication

- As long as no network contention happens this kind of work is also ideally parallelizable but has fixed execution time!

$$t_{pc} ? \frac{s_{pc}(N)}{r_{pc}} \quad \text{with} \quad u_{pc} ? 1$$

Simple Models: Critical Section

- Here we assume that all p processors try to enter the critical section at the same time. Thus the execution time scales up with p.

$$t_{cs} ? \frac{s_{cs}(N)}{r_{cs}} p \quad \text{with} \quad u_{cs} ? p$$

Simple Models: Barrier Synchronization

- There are different ways to implement a barrier synchronization. It could beimplemented in a single global instance. Then we would get a similar behaviorto the one of a critical section. It could also be implemented by using local instances and a combination operation on them. This would lead to a behavior like a broadcast communication.

$$t_{bs} ? p \quad \text{or} \quad t_{bs} ? \log(p)$$

Central resources used by all processors

- Here the same argument applies as for critical sections.

$$t_{cr} ? \frac{s_{cr}(N)}{r_{cr}} p \quad \text{with} \quad u_{cr} ? p$$

Simple Models: Overlap between communication and computation

- Here we look at the communication work which can only partially be overlapped with computational work depending on the problem size per processor. Again we use a step function ? (x) to describe whether all communication can be overlapped or not. We also assume that the amount of communication work per processor is independent of the processor number p.

$$t_{co} ? \frac{s_{co}(N)}{r_{co}} ? \begin{cases} ? \\ ? \\ ? \end{cases} N_{crit} ? \frac{f(N)}{p} \begin{cases} ? \\ ? \\ ? \end{cases}$$

Phase Modeling

- Split code in several phases.
- Based on code inspection establish algebraic timing formula. for each of these phases which may include unknown 'speed' parameter.
- Time these phases individually.
- Fit timing formulas and parameters for each phase to the measurements.

Phase Modeling

- Check for each phase the accuracy of the fit.
- Refine phases if accuracy too low.
- Timing model for complete program is the sum of all phase timing models.
- Calculate the error for the predictions.

Performance Modeling

- Some problems with analytic performance modeling are:
- Developing timing models for real applications can be very tedious.
- Possibility of systematic errors great.