

MPP Performance Optimization

Philip Mucci (mucci@cs.utk.edu)

Kevin London (london@cs.utk.edu)

University of Tennessee, Knoxville

Outline

- Performance Metrics
- Strategy
- Single Processor Optimization
- MPP Optimization
- Tools

Performance

- What is performance?
 - Execution time
 - Throughput
 - Efficiency
 - Scalability
 - Development cost
 - Maintenance Cost

Performance Examples

- Operation Weather Forecasting Model
 - Absolute execution time is key concern
 - Scalability
- Database search engine
 - Development cost and portability
- Image processing system
 - Throughput or latency

Issues in Performance

Problem Size and Precision

- Efficiency of data representation
 - Access time
 - Density
 - Memory/Disk
- Numerical representation
 - integer, float, double

Issues in Performance

Execution time

- Brute speed (MHz and bus width)
- Cycles per operation (startup + pipelined)
- Number of arithmetic units on chip
- Access to RAM (local & distributed)
- Access to disk (local & distributed)

Issues in Performance

Execution time

- Cache utilization
- Register allocation
- Instruction scheduling and pipelining
- Compiler technology
- Programming model
 - Shared memory, data parallel or SPMD.
 - Message passing or MPMD.

Parallel Performance Issues

- Compiler Optimizations
- Single node performance
- I/O and Communication
- Mapping Problem - Load Balancing
- Message Passing or Data Parallel Optimizations

Performance Metrics

- MFLOPS - Millions of floating point operations per second.
- MIPS - Millions of instructions per second.
- Execution time - Time from start to finish of our program. Also called **Wall Clock** time.

Performance Metrics

- MFLOPS/MIPS are poor measures because
 - They are dependent on the instruction set.
 - They can vary inversely to performance.
 - They say nothing about what is most important:

EXECUTION TIME

Performance Metrics

- For purposes of optimization, we are interested in:
 - Execution time of our code
 - MFLOPS of our code vs. peak rate in order to determine *efficiency*

Performance Metrics

- Fallacies
 - *MIPS is an accurate measure for comparing performance among computers.*
 - *MFLOPS is a consistent and useful measure of performance.*
 - *Synthetic benchmarks predict performance for real programs.*
 - *Peak performance tracks observed performance.*

[Hennessey and Patterson]

Performance Metrics

- Our analysis will be based upon:
 - Performance of a single machine
 - Performance of a single (*optimal*) algorithm
 - Execution time

Performance Metrics

For the purposes of comparing your codes performance among different architectures **base your comparison on time.**

...*Unless* you are completely aware of all the issues in performance analysis including architecture, instruction sets, compiler technology etc...

Asymptotic Analysis

- Algorithm X requires $O(N \log N)$ time on $O(N)$ processors
- Ignores constants and lower order terms.

$$10N > N \log N \text{ for } N < 1024$$

$$10N*N < 1000N \log N \text{ for } N < 996$$

Amdahl's Law

- The performance improvement to be gained from using some faster mode of execution is limited by the fraction of time the faster mode can be used.

$$\textit{Speedup} = \textit{Time enhanced} / \textit{Time normal}$$

$$\textit{Speedup} = \textit{Time sequential} / \textit{Time parallel}$$

- For parallel codes, there is a sequential portion $1/X$ and a parallel portion Y/X the maximum speedup is S . (∞ number of PEs)

Amdahl's Law

- Only applicable for same algorithm, starting conditions, problem size, data set, machine, etc... Fine for us.
- Problem: This ignores scaling of the problem size

Efficiency

- A measure of parallel algorithm / code quality.

$$E = \text{Time sequential} / (P * \text{Time parallel})$$

$$S = P * E$$

What is Optimization?

- Finding **hot spots & bottlenecks**
(**profiling**)
 - Code in the program that uses a *disproportional* amount of *time*
 - Code in the program that uses system resources *inefficiently*
- Reducing **wall clock** time
- Reducing resource requirements

Types of Optimization

- Hand-tuning
- Preprocessor
- Compiler
- Parallelization

Steps of Optimization

- Debug
- Profile
- Optimize blocks of code that dominate execution time
- Optimize compiler switches
- Examine correctness

Performance Strategies

- Use profiling tools before you optimize.
- Always use optimal or near optimal algorithms.
 - Be careful of requirements and problem sizes.
- The largest bottleneck should be optimized first.
- Maintain realistic and consistent input data sets/sizes during optimization.
- Know when to stop.

Performance Strategies

- The largest bottleneck should be optimized first.

PROCEDURE	TIME
<code>main ()</code>	13%
<code>procedure1 ()</code>	17%
<code>procedure2 ()</code>	20%
<code>procedure3 ()</code>	50%

- A 20% decrease of `procedure3 ()` results in 10% increase in performance.
- A 20% decrease of `main ()` results in 2.6% increase in performance

Considerations when Optimizing

Developer should be familiar with:

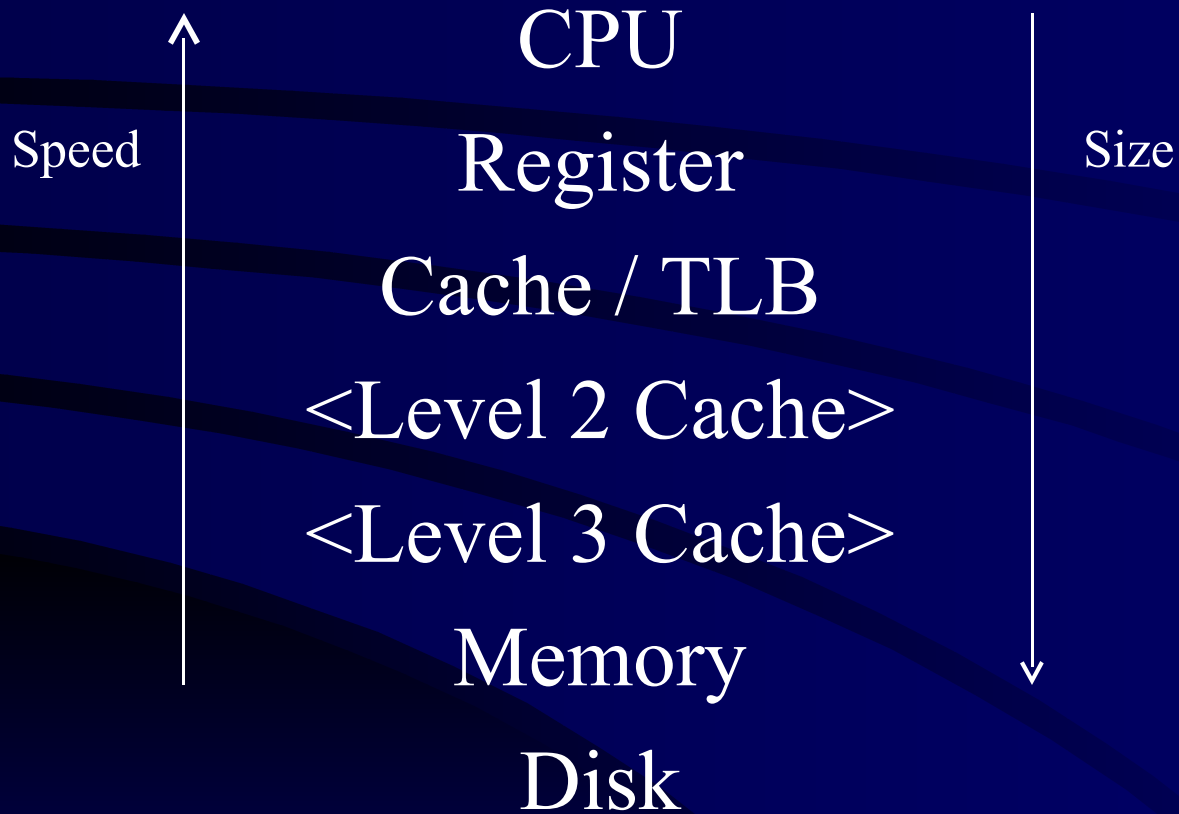
- Machine configuration, libraries and tools
- Hardware and Software overheads
- Algorithm and alternatives
- CPU/Resource requirements $O()$ notation
- Amdahl's Law
- Communication patterns
- Load balance and granularity

Locality

Spatial - If location X is being accessed, it is likely that a location *near* X will be accessed *soon*.

Temporal - If location X is being accessed, it is likely that X will be accessed again *soon*.

Memory Hierarchy



SP2 Access Times

- Register: 0 cycles
- Cache Hit: 1 cycle
- Cache Miss: 8-12 cycles
- TLB Miss: 36-56 cycles
- Page Fault: ~100,000 cycles

Cache Performance

Title:

Creator:

gnuplot

Preview:

This EPS picture was not saved
with a preview included in it.

Comment:

This EPS picture will print to a
PostScript printer, but not to
other types of printers.

Types of Cache

- Direct Mapped

One to one correspondence between memory address and cache address

- N-way Set Associative

Each memory address can live in N cache addresses, usually it's the LRU slot.

Memory Access

- Programs should be designed for maximal cache benefit.
- Minimize page faults
- Minimize large stride access to minimize TLB misses.

Memory Access

- Avoid repeatedly processing sequentially accessed data. Example for SP2: accessing 1MB of REAL*8

1MB = 244 pages * 4096 bytes;

REAL*8 = 8 bytes

4096 / 128 bytes = 32 lines/pg

128 bytes / 8 bytes = miss every 16 elements

32 misses/pg * 8 cycles/miss = 256 cycles/pg

1MB = 256 pgs

36 cycle TLB miss per page

256 * (36 + 256) = 73728 cycles wasted

Serial Optimizations

- Use most aggressive compiler options.
- Use vendor libraries.
- Improve cache utilization.
- Improve loop structure.
- Use subroutine inlining.

Array Optimization

- Stride Minimization
- Sparse Arrays
- Array Initialization
- Loop Fusion
- Floating IF's
- Loop Defactorization
- Loop Peeling
- Loop Interchange
- Loop Collapse
- Loop Unrolling
- Loop Unrolling and Sum Reduction
- Outer Loop Unrolling

Array Allocation

- Array's are allocated differently in C and FORTRAN.

1	2	3
4	5	6
7	8	9

C: 1 2 3 4 5 6 7 8 9

Fortran: 1 4 7 2 5 8 3 6 9

Array Padding

- COMMON blocks are allocated contiguously.
- Watch out for powers of two and know the associativity of your cache.
- Thrashing a 4MB direct mapped cache.

```
common /xyz/ a(1024),b(1024)
```

```
common /xyz/ a(1024),b(32),c(1024)
```

Stride Minimization

- **Stride 1**

```
for (I=0;I<1000;I++)  
    for (j=0;j<1000;j++)  
        c[I][j] += a[I][j]+b[I][j]
```

- **Stride 1000**

```
for (j=0;j<1000;j++)  
    for (i=0;i<1000;i++)  
        c[I][j] += a[i][j]+b[I][j]
```

Stride Minimization

- We must think about spatial locality.
- Effective usage of the cache provides us with the best possibility for a performance gain.
- *Recently* accessed data are likely to be faster to access.
- Tune your algorithm to minimize stride.

Stride Minimization

- Stride 1000 Loop
 - Default: 2086 ms.
 - -O3 1860 ms.
- Stride 1 Loop
 - Default: 287 ms.
 - -O3: 78 ms.

Array Initialization

- Static initialization

```
REAL(8) A(100,100) /10000*1.0/
```

- Dynamic initialization

```
DO I=1, DIM1  
  DO J=1, DIM2  
    A(I,J) = 1.0
```

Array Initialization

Which to choose?

- Static initialization requires:
 - Disk space
 - Demand paging
 - Extra Cache and TLB misses.
- Use only for small sizes with default initialization to 0.

Loop Fusion

- Loop overhead reduced
- Better instruction overlap
- Lower cache misses
- Be aware of associativity issues with array's mapping to the same cache line.

Loop Fusion

- Untuned

```
for(i=0;i<100000;i++)  
    x *= a[i] + b[i];  
for(i=0;i<100000;i++)  
    y *= a[i] + c[i];
```

- Tuned

```
for(i=0;i<100000;i++)  
{  
    x *= a[i] + b[i];  
    y *= a[i] + c[i];  
}
```


Loop Fusion

- Untuned

Default: 55 (3) ms.

-O3 39 (1.6) ms.

- Tuned

Default: 30 (6) ms.

-O3: 16 (5) ms.

Loop Interchange

- Swapping the nested order of loops
 - Minimize stride
 - Reduce loop overhead where inner loop counts are small

Loop Interchange

- Untuned

```
float a[2000][40][2];
for(i=0;i<2000;i++)
  for(j=0;j<40;j++)
    for(k=0;k<2;k++)
      a[i][j][k] *= 1.01;
```

- Tuned

```
float a[2][40][2000];
for(i=0;i<2;i++)
  for(j=0;j<40;j++)
    for(k=0;k<2000;k++)
      a[i][j][k] *= 1.01;
```

Loop Interchange

- Untuned
 - Default: 49 ms.
 - -O3: 7 ms.
- Tuned
 - Default: 48 ms.
 - -O3: 6 ms.

Floating IF's

- IF statements that do not change from iteration to iteration may be modeled out of the loop.
- Compilers can usually do this except when
 - Loops contain calls to procedures
 - Variable bounded loops that may never get entered
 - Complex loops where invariance cannot be determined

Floating IF's

- Untuned

```
for (i=0; i<1000; i++)
{
    for (j=0; j<1000; j++)
    {
        if (a[i] > 100)
            b[i] = a[i] -
3.7;
        x = x + a[j] + b[i];
    }
}
```

- Tuned

```
for (i=0; i<1000; i++)
{
    if (a[i] > 100)
        b[i] = a[i] - 3.7;
    for (j=0; j<1000; j++)
        x = x + a[j] + b[i];
}
```

Floating IF's

- Untuned
 - Default: 452 ms.
 - -O3: 120 ms.
- Tuned
 - Default: 243 ms.
 - -O3: 120 ms.

Loop Defactorization

- Loops involving multiplication by a constant in an array.
- Allows better instruction scheduling.
- Facilitates multiply-adds.

Loop Defactorization

- Untuned

```
for (i=0;i<1000;i++)
{
    a[i] = 0.0;
    for (j=0;j<1000;j++)
        a[i]+=b[j]*d[j]*c
        [i];
}
```

- Tuned

```
for (i=0;i<1000;i++)
{
    a[i] = 0.0;
    for (j=0;j<1000;j++)
        a[i]+=b[j]*d[j];
    a[i] *= c[i];
}
```

Loop Defactorization

- Note that floating point operations are not always associative.

$$(A + B) + C \neq A + (B + C)$$

- Be aware of your precision
- Always verify your results with an unoptimized code.

Loop Defactorization

- Untuned
 - Default: 270 ms.
 - -O3: 30 ms.
- Tuned
 - Default: 225 ms.
 - -O3: 28 ms.

Loop Peeling

- For loops which access previous elements in arrays. (Cylinder coordinate system)
- Compiler cannot determine that $a[jwrap]$ doesn't need to be loaded in every iteration.

Loop Peeling

- Untuned

```
jwrap = ARRAY_SIZE;
for (i=0;i<ARRAY_SIZE;i++)
{
    b[i]=(a[i]+a[jwrap])*0.5;
    jwrap = i;
}
```

- Tuned

```
b[0]=(a[0]+a[ARRAY_SIZE])*0.5;
for (i=1;i<ARRAY_SIZE;i++)
{
    b[i]=(a[i]+a[i-1])*0.5;
    jwrap = i;
}
```

Loop Peeling

- Untuned
 - Default: 26 ms.
 - -O3: 7 ms.
- Tuned
 - Default: 25 ms.
 - -O3: 6 ms.

Loop Collapse

- For multi-nested loops in which the entire array is accessed.
- This can reduce loop overhead and improve compiler vectorization.

Loop Collapse

- Untuned

```
for (i=0; i<50; i++)  
  for (j=0; j<80; j++)  
    for (k=0; k<4; k++)  
      a[i][j][k] += b[i][j]  
      [k] * c[i][j][k];
```

- Tuned

```
for (i=0; i<(50*80*4); i++)  
  a[0][0][i] += b[0][0][i]  
  * c[0][0][i];
```


Loop Collapse

- Untuned
 - Default: 5.6 ms.
 - -O3: 1.6 ms.
- Tuned
 - Default: 3.6 ms.
 - -O3: 1.4 ms.

Loop Unrolling

- Data dependence delays can be reduced or eliminated.
- Reduce loop overhead.
- Might be performed by the compiler or preprocessor. (KAP/VAST)

Loop Unrolling

Untuned

```
for (i=0; i<100; i++)  
  for (j=0; j<100; j++)  
    for (k=0; k<4; k++)  
      a[i][j] += b[k][i] * c[k][j];
```

Loop Unrolling

Tuned

```
for (i=0; i<100; i++)  
  for (j=0; j<100; j++)  
  {  
    a[i][j] += b[0][i] * c[0][j];  
    a[i][j] += b[1][i] * c[1][j];  
    a[i][j] += b[2][i] * c[2][j];  
    a[i][j] += b[3][i] * c[3][j];  
  }
```

Loop Unrolling

- Untuned
 - Default: 224 ms.
 - -O3: 25 ms.
- Tuned
 - Default: 117 ms.
 - -O3: 20 ms.

Loop Unrolling and Sum Reductions

- When an operation requires as input the result of the last output.
- Called a Data Dependency.
- Frequently happens with multi-add instruction inside of loops.
- Introduce intermediate sums. Use your registers!

Loop Unrolling and Sum Reductions

- Untuned

```
a=0.0;
for(i=0;i<ARRAY_SIZE;i++)
    for(j=0;j<ARRAY_SIZE;j++)
        a += b[j] + c[i];
```

- Tuned

```
a1 = a2 = a3 = a4 = 0.0;
for(i=0;i<ARRAY_SIZE;i++)
    for(j=0;j<ARRAY_SIZE;j+=4)
    {
        a1 += b[j] * c[i];
        a2 += b[j+1] * c[i];
        a3 += b[j+2] * c[i];
        a4 += b[j+3] * c[i];
    }
aa = a1 + a2 + a3 + a4;
```

Loop Unrolling and Sum Reductions

- Untuned
 - Default: 39 ms.
 - -O3: 12 ms.
- Tuned (2)
 - Default: 33 ms.
 - -O3: 6.1 ms.
- Tuned (4)
 - Default: 29 ms.
 - -O3: 3.6 ms.
- Tuned (8)
 - Default: 28 ms.
 - -O3: 3.0 ms.
- Tuned (16)
 - Default: 27 ms.
 - -O3: 3.3 ms.

Outer Loop Unrolling

- For nested loops, unrolling outer loop may reduce loads and stores in the inner loop.
- Compiler may perform this optimization.

Outer Loop Unrolling

- Untuned
 - Each multiply requires two loads and one store.

```
for(i=0;i<ARRAY_SIZE;i++)  
  for(j=0;j<ARRAY_SIZE;j++)  
    a[i][j] = b[i][j] * c[j];
```

Outer Loop Unrolling

- Tuned
 - Each multiply requires 5/4 loads and one store.

```
for (i=0; i<ARRAY_SIZE; i+=4)
  for (j=0; j<ARRAY_SIZE; j++)
  {
    a[i][j] = b[i][j] * c[j];
    a[i+1][j] = b[i+1][j] * c[j];
    a[i+2][j] = b[i+2][j] * c[j];
    a[i+3][j] = b[i+3][j] * c[j];
  }
```

Outer Loop Unrolling

- Untuned
 - Default: 42 ms.
 - -O3: 10 ms.
- Tuned
 - Default: 32 ms.
 - -O3: 10 ms.

Loop structure

- IF/GOTO and WHILE loops inhibit some compiler optimizations.
- Some optimizers and preprocessors can perform transforms.
- DO and for() loops are the most highly tuned.

Strength Reduction

- Reduce cost of mathematical operation with no loss in precision, compiler might do it.
- Integer multiplication/division by a constant with shift/adds
- Exponentiation by repeated multiplication
- Factorization and Horner's Rule
- Floating point division by inverse multiplication

Strength Reduction

Horner's Rule

- Polynomial expression can be rewritten as a nested factorization.

$$Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F =$$
$$(((Ax + B) * x + C) * x + D) * x + E) * x + F.$$

- Also uses multiply-add instructions
- Eases dependency analysis

Strength Reduction

Horner's Rule

- Untuned

```
for (i=0; i<10000; i++)  
{  
    x = a[i] * pow(x, 5) +  
        b[i] * pow(x, 4) +  
        c[i] * pow(x, 3) +  
        d[i] * pow(x, 2) +  
        e[i] * x + f[i];  
}
```

- Tuned

```
for (i=0; i<10000; i++)  
{  
    x = (((a[i] * x + b[i]) *  
        x + c[i]) * x + d[i]) *  
        x + e[i]) * x + f[i];  
}
```


Strength Reduction

Horner's Rule

- Untuned
 - Default: 56 ms.
 - -O3: 51 ms.
- Tuned
 - Default: 6 ms.
 - -O3: 5 ms.

Strength Reduction

Integer Division by a Power of 2

- Shift requires less cycles than division.
- Both dividend and divisor must both be unsigned or positive integers.

Strength Reduction

Integer division by a Power of 2

- Untuned

```
IL = 0
DO I=1,ARRAY_SIZE
  DO J=1,ARRAY_SIZE
    IL = IL + A(J)/2
  ENDDO
  ILL(I) = IL
ENDDO
```

- Tuned

```
IL = 0
ILL = 0
DO I=1,ARRAY_SIZE
  DO J=1,ARRAY_SIZE
    IL = IL + ISHFT(A(J),-1)
  ENDDO
  ILL(I) = IL
ENDDO
```

Strength Reduction

Integer division by a Power of 2

- Untuned
 - Default: 195 ms.
 - -O3: 37 ms.
- Tuned
 - Default: 180 ms.
 - -O3: 30 ms.

Strength Reduction Factorization

- Allows for better instruction scheduling.
- Compiler can interleave loads and ALU operations.
- Especially benefits compilers able to do software pipelining.

Strength Reduction Factorization

- Untuned

$$XX = X^*A(I) + X^*B(I) + X^*C(I) + X^*D(I)$$

- Tuned

$$XX = X^*(A(I) + B(I) + C(I) + D(I))$$

Strength Reduction Factorization

- Untuned
 - Default: 53 ms.
 - -O3: 8 ms.
- Tuned
 - Default: 49 ms.
 - -O3: 3 ms.

Subexpression Elimination Parenthesis

- Parenthesis can help the compiler recognize repeated expressions.
- Some preprocessors and aggressive compilers will do it.

Subexpression Elimination Parenthesis

- Untuned

$$XX = XX + X(I) * Y(I) + Z(I) + X(I) * Y(I) - Z(I) + X(I) * Y(I) + Z(I)$$

- Tuned

$$XX = XX + (X(I) * Y(I) + Z(I)) + X(I) * Y(I) - Z(I) + (X(I) * Y(I) + Z(I))$$

Subexpression Elimination Parenthesis

- Untuned
 - Default: 38 ms.
 - -O3: 25 ms.
- Tuned
 - Default: 33 ms.
 - -O3: 20 ms.

Subexpression Elimination

Type Considerations

- Changes the type or precision of data.
 - Reduces resource requirements.
 - Avoid type conversions.
 - Processor specific performance.
- Example convert `REAL*16` to `REAL*8`

Subexpression Elimination

Type Considerations

- Untuned
 - Default: 157 ms.
 - -O3: 129 ms.
- Tuned
 - Default: 35 ms.
 - -O3: 13 ms.

I/O Considerations

- Memory map files.
- Match record size near multiple of cache line size.
- Use near power of two record size.

I/O Considerations

- I/O is orders of magnitude slower than memory access.
- Eliminate unnecessary I/O.
- Use best suited interface.
- Avoid copies.
- Use binary I/O

Optimized Arithmetic Libraries

- [P]BLAS: Basic Linear Algebra Subroutines
- [Sca]LAPACK: Linear Algebra Package
- ESSL: Engineering and Scientific Subroutine Library
- NAG: Numerical Algorithms Group
- IMSL: International Mathematical and Statistical Lib.
- Available for Fortran, C, C++ etc...

Optimized Arithmetic Libraries

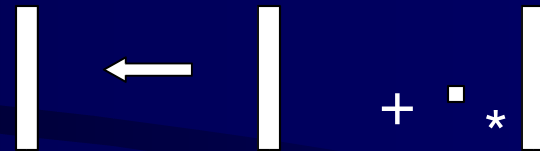
- Advantages:
 - Subroutines are quick to code and understand.
 - Routines provide *portability*.
 - Routines perform well.
 - Comprehensive set of routines.

BLAS

- Common Matrix/Matrix, Matrix-Vector, Vector-Vector.
- REAL/DOUBLE/COMPLEX
- Reference version available from UT.
- Vendor version offer high performance.
- Multithreaded are sometimes available.
- <http://www.netlib.org/blas/index.html>

Level 1, 2 and 3 BLAS

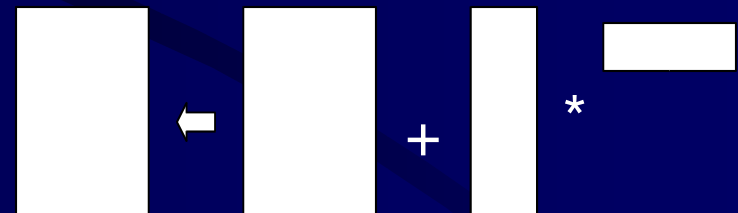
- Level 1 BLAS
Vector-Vector
operations



- Level 2 BLAS
Matrix-Vector
operations

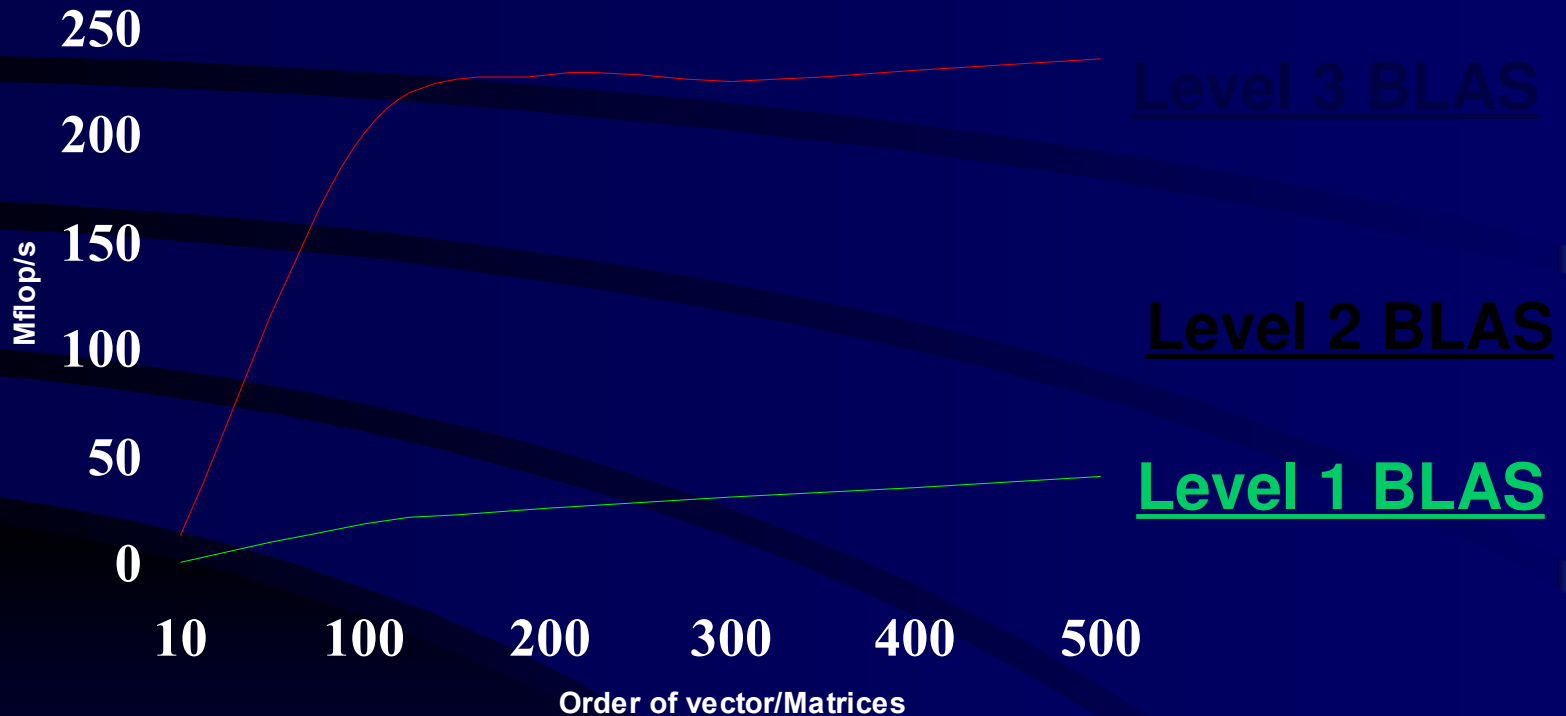


- Level 3 BLAS
Matrix-Matrix
operations



BLAS for Performance

IBM RS/6000-590 (66 MHz, 264 Mflop/s Peak)



- Development of blocked algorithms important for performance

BLAS Performance

Title:

Creator:

gnuplot

Preview:

This EPS picture was not saved
with a preview included in it.

Comment:

This EPS picture will print to a
PostScript printer, but not to
other types of printers.

BLACS -- Introduction

- A **design tool**, they are a conceptual aid in design and coding.
- Associate widely recognized mnemonic names with communication operations, improve
 - **program readability**,
 - **self-documenting quality** of the code.
- Promote **efficiency** by identifying frequently occurring operations of linear algebra which can be optimized on various computers.

BLACS -- Basics

- An operation which involves more than one sender and one receiver is called a “scoped operation”.
Using a 2D-grid, there are 3 natural scopes:

Scope	Meaning
Row	All processes in a process row participate.
Column	All processes in a process column participate.
All	All processes in the process grid participate.

BLACS -- Basics

- Operations are **matrix-based** and **ID-less**
- Types of BLACS routines: **point-to-point** communication, **broadcast**, **combine** operations and **support** routines.

PBLAS -- Introduction

- **Parallel Basic Linear Algebra Subprograms** for distributed-memory MIMD computers
- Do both the communication and computation.
- **Simplification of the parallelization:** especially when BLAS-based,
- **Modularity:** gives programmer larger building blocks,
- **Portability:** machine dependencies are confined to the BLAS and BLACS.

Scope of the PBLAS

- No vector rotations,
- No dedicated subprograms for banded matrices,
- Matrix transposition available.
- Prototype version of packed storage PBLAS
 - <http://www.netlib.org/scalapack/prototype/>

PBLAS

- Similar to the BLAS in functionality and naming.
- Built on the BLAS and BLACS
- Provide global view of matrix

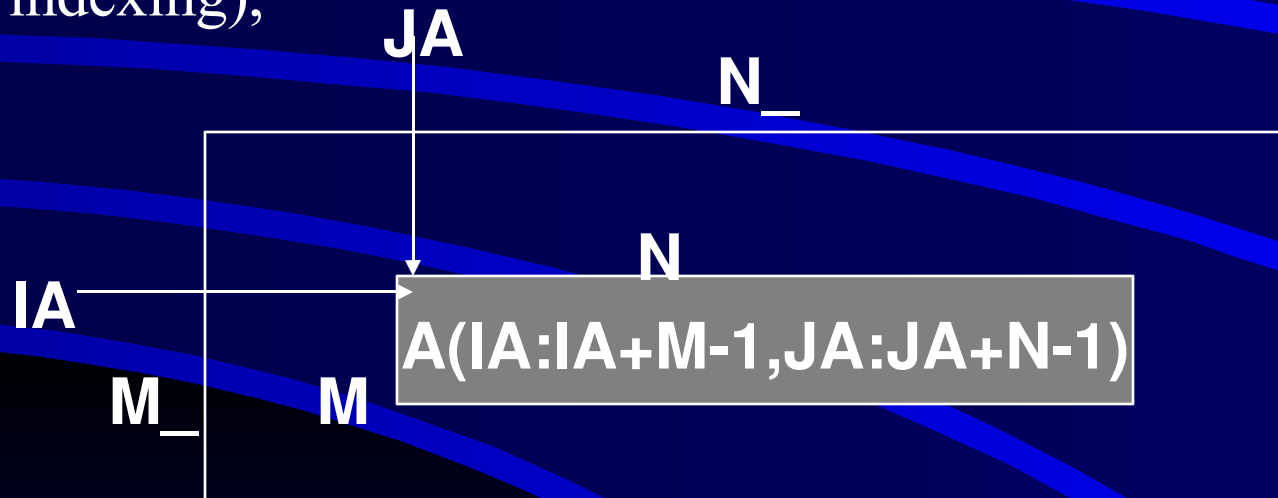
```
CALL DGEXX ( M, N, A( IA, JA ), LDA,... )
```




```
CALL PDGEXX( M, N, A, IA, JA, DESCA,... )
```

PBLAS -- Syntax

- **Global view of the matrix operands**, allowing global addressing of distributed matrices (hiding complex local indexing),



PBLAS -- Storage Conventions

- An $M_$ -by- $N_$ matrix is **block-partitioned** into $MB_$ -by- $NB_$ blocks and distributed according to the **two-dimensional block-cyclic scheme**
 **load balanced computations, scalability**
- Locally, the scattered **columns are stored contiguously** (FORTRAN “Column Major”)
 - **re-use of the BLAS** (leading dimension $LLD_$)

PBLAS -- Examples

```
INTEGER  IAM, ICTXT, INFO, LDA, LDB, LDC, NMAX, NPROCS
PARAMETER ( NMAX = 3, LDA = NMAX, LDB = NMAX, LDC = NMAX )
INTEGER  DESCA( 9 ), DESCB( 9 ), DESCC( 9 )
DOUBLE PRECISION  A( NMAX, NMAX ), B( NMAX, NMAX ), C( NMAX,
    NMAX )

CALL BLACS_PINFO( IAM, NPROCS )
IF( NPROCS.LT.1 ) THEN
    NPROCS = 4
    CALL BLACS_SETUP( IAM, NPROCS )
END IF
CALL BLACS_GET( -1, 0, ICTXT )
CALL BLACS_GRIDINIT( ICTXT, 'Row', 2, 2 )
```

PBLAS -- Examples

...

```
CALL DESCINIT( DESCA, 5, 5, 2, 2, 0, 0, ICTXT, LDA, INFO )
```

```
CALL DESCINIT( DESCB, 5, 5, 2, 2, 0, 0, ICTXT, LDB, INFO )
```

```
CALL DESCINIT( DESCC, 5, 5, 2, 2, 0, 0, ICTXT, LDC, INFO )
```

...

```
CALL PDGEMM( 'No transpose', 'No transpose', 4, 4, 4, 1.0D+0,  
$           A, 1, 1, DESCA, B, 1, 1, DESCB, 0.0D+0, C, 1, 1, DESCC )
```

...

```
CALL PBFREEBUF()
```

```
CALL BLACS_GRIDEXIT( ICTXT )
```

```
CALL BLACS_EXIT( 0 )
```

Features of PBLAS V2 ALPHA

- Software is backward compatible with current version 1.5.
- Improved ease-of-use:
 - Previous alignment restrictions have all been removed. Re-alignment is performed on-the-fly and only when necessary.
 - General rectangular block cyclic distribution of the operands is supported for all routines.

Features of PBLAS V2 ALPHA

- Support for replicated operands
 - Increased functionality is truly usable as illustrated by the packed storage ScaLAPACK prototype routines
 - Enable the expression of currently missing algorithms in the ScaLAPACK dense library such as constrained linear least squares solvers.
 - Simply the expression of complex algorithms such as divide and conquer algorithms (sign function).

LAPACK

- F77 routines for solving
 - systems of simultaneous linear equations and eigenvalue problems
 - matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur)
 - Related computations such as reordering and conditioning.
 - Built on the level 1, 2 3 BLAS Single, Double, Complex, Double Complex
- <http://www.netlib.org/lapack/index.html>

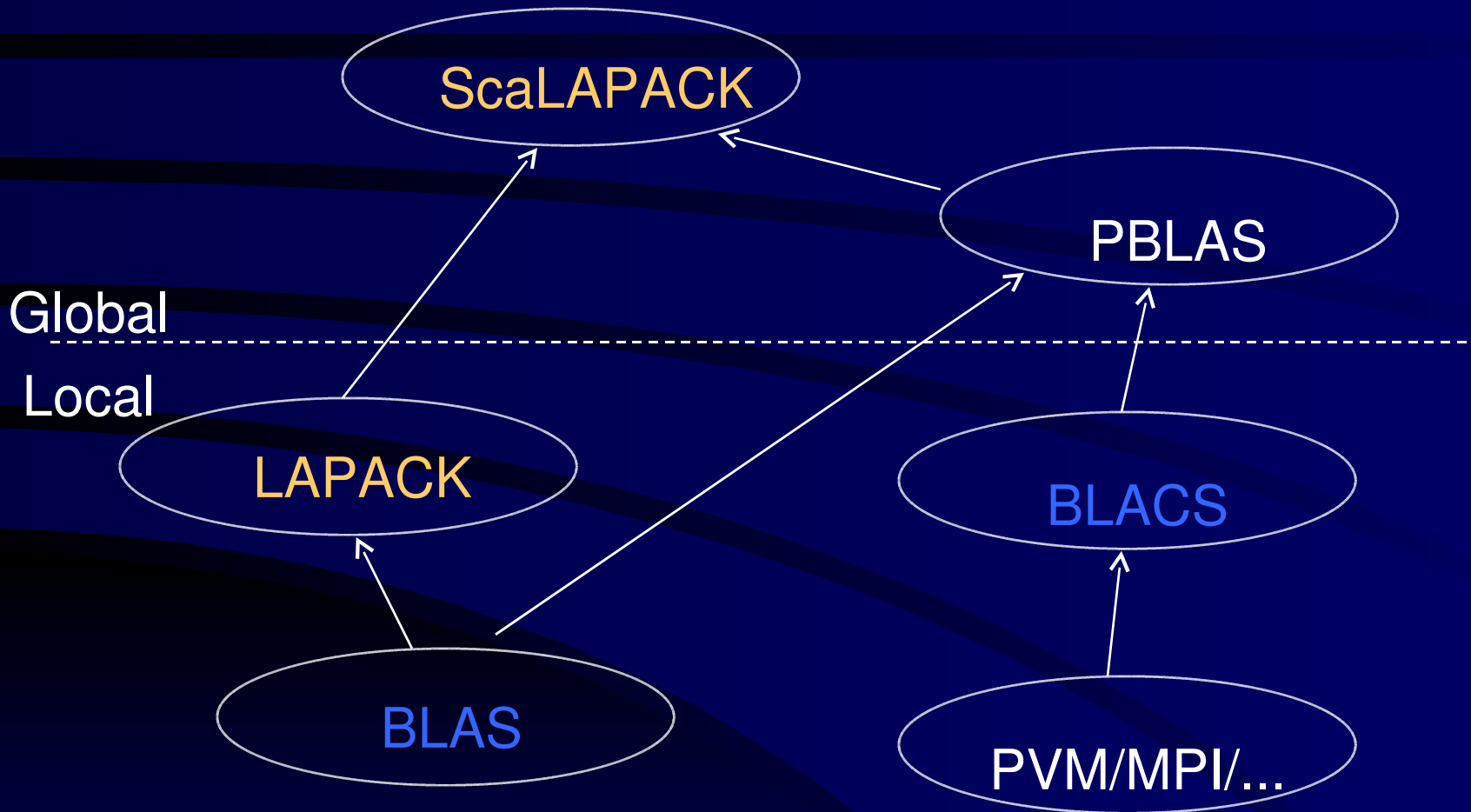
LAPACK

- Most of the parallelism in the BLAS.
- Advantages of using the BLAS for parallelism:
 - Clarity
 - Modularity
 - Performance
 - Portability

LAPACK -- Release 3.0

- Add functionality
 - divide and conquer SVD,
 - error bounds for GLM and LSE,
 - new expert drivers for GSEP,
 - faster QRP,
 - faster solver for the rank-deficient LS (xGELSY),
 - divide and conquer least squares
 - ...

ScaLAPACK Structure



LAPACK - Goals

- **Efficiency**
 - Optimized compute and communication engines
 - Block-partitioned algorithms (Level 3 BLAS) utilize memory hierarchy and yield good node performance
- **Scalability**
 - as the problem size and number of processors grow
- **Reliability**
 - Whenever possible, use LAPACK algorithms and error bounds.
- **Portability**
 - isolate machine dependencies to BLAS and the BLACS
- **Flexibility**
 - Modularity: Build rich set of linear algebra tools: BLAS, BLACS, PBLAS
- **Ease-of-Use**
 - Calling interface similar to LAPACK

ScaLAPACK Implementation

- BLAS (Performance and Portability)
 - Blocked data access (Level 3 BLAS) yields good local performance,
 - Portability: standardization of efficient kernels.
- BLACS (Performance and Portability)
 - Correct level of notation: communication of matrices,
 - Efficiency: identify frequent linear algebra operations which can be optimized on various computers.

ScaLAPCK Functionality

Problem type	SDrv	EDrv	Factor	Solve	Inv	Cond Est	Iter Refin
$Ax = b$							
Triangular				X	X	X	X
SPD	X	X	X	X	X	X	X
SPD Banded	X		X	X			
SPD Tridiagonal	X		X	X			
General	X	X	X	X	X	X	X
General Banded	X		X	X			
General Tridiagonal	X		X	X			
Least squares	X		X	X			
GQR			X				
GRQ			X				
$Ax = \lambda x$ or $Ax = \lambda Bx$	SDrv	Edrv	Reduct	Solution			
Symmetric	X	X	X	X			
General	+		X	+			
Generalized BSPD			X	X			

- Timing and Testing routines for almost all
- This is a large component of the package
- Prebuilt libraries available for SP, PCA, O2K, PGON, ALPHA, HPPA, LINUX, Sun, RS6K

ScaLAPACK Functionality

- Orthogonal/unitary transformation routines
- Prototype Codes
 - PBLAS (version 2.0 ALPHA)
 - Packed Storage routines for LLT, SEP, GSEP
 - Out-of-Core Linear Solvers for LU, LLT, and QR
 - Matrix Sign Function for Eigenproblems
 - SuperLU and SuperLU_MT
 - HPF Interface to ScaLAPACK

Parallelism in ScaLAPACK

- Level 3 BLAS block operations
 - All the reduction routines
- Pipelining
 - QR Algorithm, Triangular Solvers, classic factorizations
- Redundant computations
 - Condition estimators
- Static work assignment
 - Bisection
- Task parallelism
 - Sign function eigenvalue computations
- Divide and Conquer
 - Tridiagonal and band solvers, symmetric eigenvalue problem and Sign function
- Cyclic reduction
 - Reduced system in the band solver

Narrow Band and Tridiagonal Matrices

- The ScaLAPACK routines solving **narrow-band** and **tridiagonal** linear systems assume
 - the narrow band or tridiagonal coefficient matrix to be distributed in a block-column fashion, and
 - the dense matrix of right-hand-side vectors to be distributed in a block-row fashion.
- Divide-and-conquer algorithms have been implemented because they offer greater scope for exploiting parallelism than the corresponding adapted dense algorithms.

ScaLAPACK Documentation

- Documentation
 - ScaLAPACK Users' Guide
http://www.netlib.org/scalapack/slug/scalapack_slug.html
 - Installation Guide for ScaLAPACK
 - LAPACK Working Notes
- Test Suites for ScaLAPACK, PBLAS, BLACS
- Example Programs
<http://www.netlib.org/scalapack/examples/>
- Prebuilt ScaLAPACK libraries on netlib

ESSL

- provides 400 functions:
 - Basic Linear Algebra Subroutines (BLAS)
 - Linear System of Equations
 - Eigensystem Analysis
 - Fourier Transforms
- <http://www.tc.cornell.edu/UserDoc/Software/Num/essl/>

Optimized Arithmetic Libraries

- NAG - Routines for solving
 - Minimization, integral equations
 - ODE's, PDE's, transforms, quadrature
 - Linear algebra, nonlinear equations
 - Curve, surface fitting, smoothing
 - Statistics and estimation
 - Multivariate, time-series, tabular and contingency analysis
 - <http://www.nag.com>

Optimized Arithmetic Libraries

- IMSL - Over 900 routines for solving common mathematical and statistical tasks.
- NAG is better for optimization, IMSL better for statistics.
- <http://www.vni.com/products/imsl/fortfunc.html>

Parallel Optimization

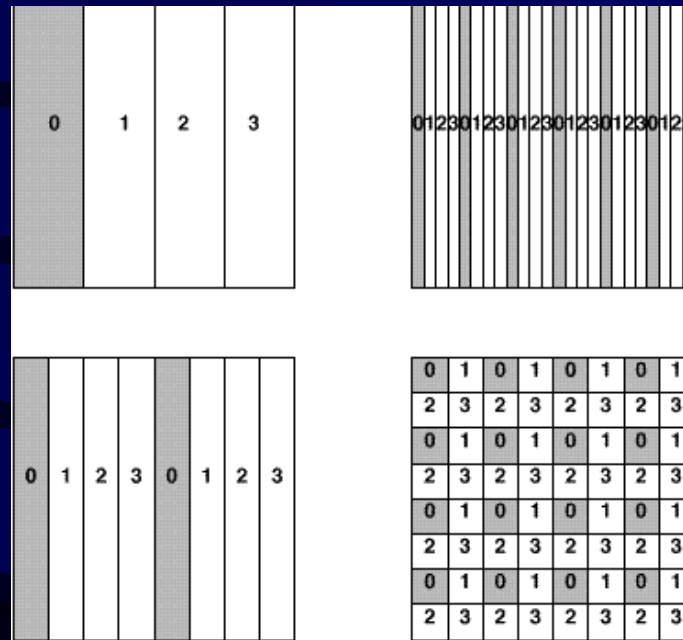
- Two programming models.
 - Message Passing
 - Shared Memory
- Optimizing parallel code

Choosing a Data Distribution

- The two main issues in choosing a data layout for dense matrix computations are:
 - **load balance**, or splitting the work reasonably evenly among the processors throughout the algorithm, and
 - **use of the Level 3 BLAS** during computations on a single processor to utilize the memory hierarchy on each processor.

Possible Data Layouts

- 1D block and cyclic column distributions



- 1D block-cyclic column and 2D block-cyclic distribution used in ScaLAPACK

Two-dimensional Block-Cyclic Distribution

- **Ensure good load balance --> Performance and scalability,**
- Encompasses a large number of (but not all) data distribution schemes,
- Need redistribution routines to go from one distribution to the other.

Load Balancing

- Static
 - Data/tasks are partitioned among existing processors.
 - Problem of finding an efficient mapping
- Dynamic
 - Master/Worker model
 - Synchronization and data distribution problems

MPP Optimization

- Programming
 - Message passing (MPI, PVM, Shmem)
 - Shared memory (HPF or MP directive based)
- Algorithms
 - Data or Functional Parallelism
 - SIMD, MIMD
 - Granularity (fine, medium, coarse)
 - Master/Worker or Hostless

Parallel Performance

- Architecture is characterized by
 - Number of CPU's
 - Connectivity
 - I/O capability
 - Single processor performance

Message Passing

- Two popular message passing API's.
 - PVM
 - UT/ORNL
 - Vendor
 - MPI
 - MPICH from MS State
 - LAM from Ohio Supercomputing Center
 - Vendor

Message Passing

- In general
 - PVM is a message passing research vehicle.
 - MPI is a production product intended for application engineers.
 - MPI will outperform PVM.
 - MPI has richer functionality
 - PVM is better for applications requiring fault tolerance, heterogeneity and changing number of processes.

Message Passing

- Both PVM and MPI
 - Support collective operations
 - Support customized data types
 - Will take advantage of shared memory
 - Exist on almost every platform including
 - Networks of workstations
 - Windows 95 and NT
 - Multiprocessor workstations

Message Passing

- Node 1 needs X bytes from node 0
- Node 0 calls a send function (X bytes from address A)
- Node 1 calls a receive function (X bytes into address B)

Message Passing

- Upon message arrival
 - If node B has not **posted** a receive the data is **buffered** until the receive function is called.
 - Else the data is copied directly to the address given to the receive function.

Communication Issues

- Startup time, latency or overhead
- Bandwidth
- Network contention and congestion
- Bidirectionality
- Communication API
- Dedicated Channels

Communication Issues

- Startup time and bandwidth
 - Startup time is higher than the time to actually transfer a *small* message.
 - Send larger messages fewer times, but try to keep everyone busy.
- Contention can be reduced by uniformly distributing messages.

Communication Issues

- To take advantage of bidirectionality, post receives before sending.
- Dedicated channels
 - On the SP2 make sure you use the User-Space communication option. This will double your bandwidth and half your latency.
 - `setenv MP_EUILIB us`
- As mentioned, use `MPI_Ixxx` calls.
 - It can handle more particles than fit in memory

Message Passing

Buffering - Temporary storage of data.

Posting - Temporary storage of an address.

Nonblocking - Refers to an function A that initiates an operation B and returns to the caller before the completion of B.

Blocking - The function A does not return to the caller until the completion of operation B.

Polling/Waiting - Testing for the completion of a nonblocking operation.

MPI Bandwidth

Title:

Creator:

gnuplot

Preview:

This EPS picture was not saved
with a preview included in it.

Comment:

This EPS picture will print to a
PostScript printer, but not to
other types of printers.

MPI Send Latency

Title:

Creator:

gnuplot

Preview:

This EPS picture was not saved
with a preview included in it.

Comment:

This EPS picture will print to a
PostScript printer, but not to
other types of printers.

Message Passing

- It is possible for sends and receives to be
 - Nonblocking(send) or Posted(receive)
 - Synchronous(send)
 - Buffered
 - Blocking

PVM Message Passing

- In PVM, all transfers are buffered except for some MPP and vendor implementations.
- All sends complete when it is safe to use the input argument again.
(transmitted/buffered)
- Transmission does not guarantee buffering or completion of the sent data by the remote process.

PVM Optimizations

- We are primarily interested in

`PVM_PSEND` `PVM_PRECV`

- Why? Because it avoids the cost of memory allocation, packing and data translation.
- In addition, `PVM_PRECV` allows in-place receives on MPP's.

PVM Optimizations

- Avoid packing data, this implies an extra copy and data translation. By default PVM uses XDR encoding.
- If you must pack a message, use
`pvm_initsend(PvmDataInPlace)`
- If communicating off your machine, bypass the PVM daemon with
`pvm_setopt(PvmRoute, PvmRouteDirect)`

PVM Optimizations

- Avoid use of the group server, roll your own collective operations.

```
pvm_joingroup()
```

```
pvm_bcast()
```

```
pvm_barrier()
```

```
pvm_reduce()
```

```
pvm_gsize()
```

```
pvm_getinst()
```

```
pvm_lvgroup()
```

MPI Message Passing

- MPI introduces communication **modes** dictating semantics of completion of send operations.
 - **Buffered** - When transmitted or buffered, space provided/limited by application, else error.
 - **Ready** - Only if receive is posted, else error.
 - **Synchronous** - Only when receive begins to execute, else wait. Useful for debugging.

MPI Message Passing

- In addition

standard - MPI will decide if/how much outgoing data is buffered. If space is unavailable, completion will be delayed until data is transmitted to receiver. (Like PVM)

Immediate - nonblocking, returns to the caller ASAP. May be used with any of the above modes.

MPI Message Passing

- Ready sends can remove a handshake for large messages.
- There is only one receive mode, it matches any of the send modes.

MPI Optimizations

- We are primarily interested in
`MPI_ISEND`, `MPI_IRECV`, `MPI_IRSEND`
- Why? Because your program could be doing something useful while sending or receiving! You can hide much of the cost of these communication operations.

MPI Data Types

- For array transfers MPI has user defined data types to gather and scatter data to/from memory.
- Try to use `MPI_TYPE_[H]VECTOR()` or `MPI_TYPE_[H]INDEXED()`
- **Avoid** `MPI_TYPE_STRUCT()`

MPI Collective Communication

- Unlike PVM, with MPI you should use the collective operations. They are likely to be highly tuned for the architecture.
- These operations are very difficult to optimize and are often the bottlenecks in parallel applications.

MPI Collective Communication

`MPI_Barrier()`

`MPI_Bcast()`

`MPI_Gather[v]() MPI_Scatter[v]()`

`MPI_Allgather[v]()`

`MPI_Alltoall[v]()`

`MPI_Reduce()`

`MPI_AllReduce()`

`MPI_Reduce_Scatter()`

`MPI_Scan()`

Message Passing Optimizations

- Try to keep message sizes *not small*
- Try to pipeline communication/computation
- Avoid data translation and data types unless necessary for good performance
- Avoid wildcard receives
- Align application buffers to double words and page sizes. Be careful of cache lines!

Message Passing Optimization

Nearest Neighbor Example 1

N slave processors available plus Master, M particles each having (x,y,z) coordinates.

- 1) Master reads and distributes all coordinates to N processors.
- 2) Each processor calculates its subset of M/N and sends it back to the master.
- 3) Master processor receives and outputs information.

Message Passing Optimization

Nearest Neighbor Example 2

- 1) Master reads and scatters M/N coordinates to N processors.
- 2) Each processor receives its own subset and makes a replica.
- 3) Each processor calculates its subset of M/N coordinates versus the replica.
- 4) Each processor sends to the next processor its replica of M/N coordinates.
- 5) Each processor receives the replica. Goto 3) $N-1$ times.
- 6) Each processor sends its info back to the Master

Message Passing Optimization

Nearest Neighbor Example

- Example 1 works better only when
 - There are a small number of particles
 - You have an super efficient broadcast
- Example 2 works better more often because
 - Computation is pipelined. Note that slave processor 0 is already busy before processor 1 even gets its input data.

MPI Message Passing

- To test for the completion of a message use `MPI_WAITxxx` and `MPI_TESTxxx` where `xxx` is `all`, `any`, `some` or `NULL`.
- Remember you must test `ISEND`'s as well as `IRECV`'s before you can reuse the argument.

Automatic Parallelization

- Let the compiler do the work.
- Advantages
 - It's easy
- Disadvantages
 - Only does loop level parallelism.
 - It wants to parallelize every loop iteration in your code.

Automatic Parallelization

- On the SGI

```
f77 -pfa <prog.f>
```

- Tries to parallelize every loop in your code.

Data Parallelism

- **Data parallelism:** different processors running the same code on different data. (SPMD)
- Identify hot spots.
- Do it by hand via directives.
- Modify the code to remove dependencies.
- Make sure you get the right answers.

Data Parallelism on the SGI's

- Insert the `c$doacross` directive just before the loop to be parallelized.
- Declare local and shared variables
- Compile with `-mp` option.

```
c$doacross local(i) share(a,n)
  do i=1,n
    a(i)=float(i)
  end do
```

Data Parallelism on the SGI's

- Directives affect only immediately referenced loop.
- Directives begin in column one.
- `c$doacross` is becoming a standard.

Data Parallelism on the SGI's

- Compiler generates code that runs with any number of threads settable at runtime.
- Set number of threads.

```
pagh> setenv MP_SET_NUMTHREADS 4
```

Task Parallelism

- **Task parallelism** means different processors are running different procedures.
- Can be accomplished on *any* machine with data parallel directives via if statements inside a loop.

Task Parallelism

```
c$doacross local(i)
  do i=1,n
    if (i=1) call sub1(...)
    if (i=2) call sub2(...)
    if (i=3) call sub3(...)
    if (i=4) call sub4(...)
  end do
```

Limits on Parallel Speedup

- The code is I/O bound.
- The problem size is fixed.
- The problem size is too small.
- There is too much serial/scalar code.
- The algorithm is inherently serial.
- Data distribution.
- Parallel overhead.

Parallel Overhead

- Creating/Scheduling threads
- Communication
- Synchronization
- Partitioning

Parallel Overhead

- For data parallel programming we can estimate parallel overhead.
- Time the code with only one thread

Reducing Parallel Overhead

- Don't parallelize ALL the loops.
- Don't parallelize the small loops.
- Use the “if” modifier.

```
c$doacross if(n > 500), local(...), share(...)  
do i=1,n  
enddo
```

Reducing Parallel Overhead

- Use task parallelism.
 - Lower overhead
 - More code runs in parallel
 - Requires a parallel algorithm

Improving Load Balance

- Change the way loop iterations are allocated to threads.
 - Change the scheduling type
 - Change the chunk size

Improving Load Balance

- Scheduling

- `setenv MP_SCHEDULETYPE <type>`
- `c$doacross mp_schedtype=<type>`
- **SIMPLE** - default, iterations equally and sequentially allocated per processor.
- **INTERLEAVE** - round-robin per chunk of iterations. Use when some iterations do more work than others.

Improving Load Balance

- Scheduling
 - DYNAMIC - iterations are allocated per processor during run-time. When the amount of work is unknown.
 - GSS - guided self scheduling. Each processor starts with a large number and finishes with a small number.

Improving Load Balance

- Change the number of iterations performed per processor.
 - `setenv CHUNK 4`
 - `c$doacross local(i) chunk_size=4`

SGI Origin 2000

- MIPS R10000, 195Mhz, 5.1ns
- 64 Integer, 64 Floating Point Registers
- 4 Instructions per cycle
- Up to 2 Integer, 2 Floating Point, 1 Load/Store per cycle
- 4 outstanding cache misses
- Out of order execution

SGI Origin 2000

- 64 Entry TLB, variable page size
- 32K Data, 32K Instruction, 4MB unified.
- 2-way set associative, LRU replacement, 2-way interleaved, except instruction cache.
- 128B line size.
- 624MB/sec CrayLink interconnect.

SP2

- IBM Power 2 SC, 135Mhz
- 32 Integer, 32 Floating Point Registers
- 6(8) Instructions per cycle
- 2 Integer, 2 Floating Point, 1 Branch,. 1 Conditional
- Zero cycle branches, dual FMA

SP2

- 256 Entry TLB
- Primary Cache 128K Data, 32K Instruction
- 4 way set associative
- 256B line size
- 150MB/sec interconnect

T3E

- Alpha 21164, 450Mhz
- Primary Cache 8K Data, 8K Instruction
- 96KB on-chip 3-way associative secondary cache
- 2 FP / 2 Int / cycle

T3E

- Scheduling very important
- 64 bit divides take 22-60 CP
- Ind Mult/Add takes 4 cp, but issued every cycle

T3E

- Latency hiding features
 - Cache bypass
 - Streams
 - E-registers
- 6 queued Dcache misses/WBs to Scache
- Load/store mergeing
- 32/64 byte line Dcache/Scache
- 2 cycle/8-10 cycle hit Dcache/Scache

T3E Cache Bypass

- Reduces memory traffic requirements.

- Fortran

```
!DIR$ CACHE_BYPASS var1, var2
```

- C

```
#pragma _CRI cache_bypass var1,  
var2
```

- Block copy

– 593 MB/sec vs 401 MB/sec

T3E Streams

- Designed to provide automatic prefetching for densely strided data.
- 6 stream buffers, two 64 byte lines each
- Starts when 2 contiguous misses
- Look at difference in loads
 - 875MB/sec with streams
 - 296MB/sec without

T3E Streams

- Count references to memory in your loops, make sure no more than six.
- Stores take up streams due to WB nature of Scache.
- May need to split loops to reduce streams.
- To use them

```
setenv SCACHE_D_STREAMS 1  
man intro_streams  
man streams_guide
```

T3E E-registers

- 512 64-bit off-chip registers for transferring data to/from remote/local memory
- SHMEM library
 - Shared, distributed, memory access routines that use the E-registers.
 - Can work on local memory
- `man intro_shmem`
- Block copy
 - 775 MB/sec vs 401 MB/sec

T3E E-registers

- Benchlib library
 - One sided data transfers from memory to E-registers
 - More complicated to use than SHMEM
 - Not supported by Cray
 - Nonblocking
 - Scatter / Gather
- Block copy
 - 592 MB/sec vs 401 MB/sec

T3E/SGI Software Pipelining

- Allows mixing of iterations from different loops in each iteration of the hardware loop
- More work per cycle
- Not as important for dynamically scheduled processors. It is turned on by default when you use `-O3` on the R8000.
- Use `-pipeline2` on the T3E.

Subroutine Inlining

- Replaces a subroutine call with the function itself.
- Useful in loops that have a huge iteration count.
- Allows parallelization.

O2K Flags and Libraries

- O, -O2 - Optimize
- O3 - Maximal generic optimization, may alter semantics.
- Ofast=ip27 - SGI compiler group's best set of flags.
- IPA=on - Enable interprocedural analysis.
- n32 - 32-bit object, best performer.
- copt - Enable the C source-to-source optimizer.
- INLINE:<func1>,<func2> - Inline all calls to func1 and func2.
- LNO - Enable the loop nest optimizer.
- cord - Enable reordering of instructions based on feedback information.
- feedback - Record information about the programs execution behavior to be used by IPA, LNO and -cord.
- lcomplib.sgimath -lfastm - Include BLAS, FFTs, Convolutions, EISPACK, LINPACK, LAPACK, Sparse Solvers and the fast math library.

SP2 Flags and Libraries

- O, -O2 - Optimize
- O3 - Maximum optimization, may alter semantics.
- qarch=pwr2, -qtune=pwr2 - Tune for Power2.
- qcache=size=128k,line=256 - Tune Cache for Power2SC.
- qstrict - Turn off semantic altering optimizations.
- qhot - Turn on addition loop and memory optimization, Fortran only.
- Pv, -Pv! - Invoke the VAST preprocessor before compiling. (C)
- Pk, -Pk! - Invoke the KAP preprocessor before compiling. (C)
- qhsflt - Don't round floating floating point numbers and don't range check floating point to integer conversions.
- inline=<func1>,<func2> - Inline all calls to func1 and func2.
- qalign=4k - Align large arrays and structures to a 4k boundary.
- lesslp2 - Link in the Engineering and Scientific Subroutine Library.

T3E Flags and Libraries

- O, -O2 - Optimize
- O3 - Maximum optimization, may alter semantics.
- apad - Pad arrays to avoid cache line conflicts
- unroll2 - Apply aggressive unrolling
- pipeline2 - Software pipelining
- split2 - Apply loop splitting.
- Wl"-Dallocate(alignsz)=64b" Align common blocks on cache line boundary
- lmfastv - Fastest vectorized intrinsics library
- lsci - Include library with BLAS, LAPACK and ESSL routines
- inlinefrom=<> - Specifies source file or directory of functions to inline
- inline2 - Aggressively inline function calls.

Timers

- `time <command>` returns 3 kinds.
 - Real time: Time from start to finish
 - User: CPU time spent executing your code
 - System: CPU time spent executing system calls
- `time` on the SGI.
- Warning! The definition of CPU time is different on different machines.

Timers

- Sample output for csh users:

```
1          2          3          4          5          6          7
1.150u  0.020s  0:01.76  66.4  15+3981k  24+10io  0pf+0w
```

1) User (ksh)

2) System (ksh)

3) Real (ksh)

4) Percent of time spent on behalf of this process, not including waiting.

5) 15K shared, 3981K unshared

6) 24 input, 10 output operations

7) No page faults, no swaps.

Timers

- `gettimeofday()` , part of the C library obtains seconds and microseconds since Jan 1, 1970.
- Resolution is hardware dependent, near microsecond for SP2, T3E and SGIs.
- Latency is not the same as resolution.
 - Many calls to this function will affect wall clock time.

Timers

- `mclock()` returns machine *ticks* as `REAL*4` of the current process. *Includes all children*
- SP2 also has `rtc()` and `irtc()` taken directly from hardware.
- `rtc()` returns `REAL*8` of seconds since initial value.
- `irtc()` returns `INTEGER*8` of nanoseconds since the initial value.

Timers

- T3E has RTC() which returns cycle counter as REAL. Use the following to get speed in seconds of a cycle.

```
float CLOCKTICK(void) {  
    long sysconf(int request);  
    float p;  
    p = (float)sysconf(_SC_CRAY_CPCYCLE);  
    p *= 1.0e-12;  
    return(p); }
```


Timers

- `MPI_Wtime()` returns elapsed wall clock time in seconds as a double.
- This is portably one of the most efficient timers. We can use it for serial programs.

C

```
double start;  
MPI_Init(&argc, &argv);  
start = MPI_Wtime();  
MPI_Finalize();
```

Fortran

```
integer ierr  
double start  
call MPI_INIT(ierr);  
start = MPI_WTIME();  
call MPI_FINALIZE(ierr)
```

prof

- Profiles program execution at the procedure level
- Available on most Unix systems, not T3E
- Displays the following:
 - Name, percentage of CPU time
 - Cumulative and average execution time
 - Number of time procedure was called

prof

- Compile your code with `-p`
- After execution the CWD will contain `mon.out.[x]`
- Type `prof`, it will look for `mon.out` in the CWD. Otherwise give it name(s) with the `-m option`
- Format of output is:

```
Name %Time Seconds Sumsecs #Calls msec/call
```

gprof

- Profiles programs according to their call graphs
- Available on most Unix systems, not T3E
- Information different from `prof`:
 - Adds the parent of each procedure
 - Adds an index number for each procedure
 - Adds direct descendents of each procedure
 - Adds breakdown of time used by descendents
 - Percentage of CPU time is cumulative

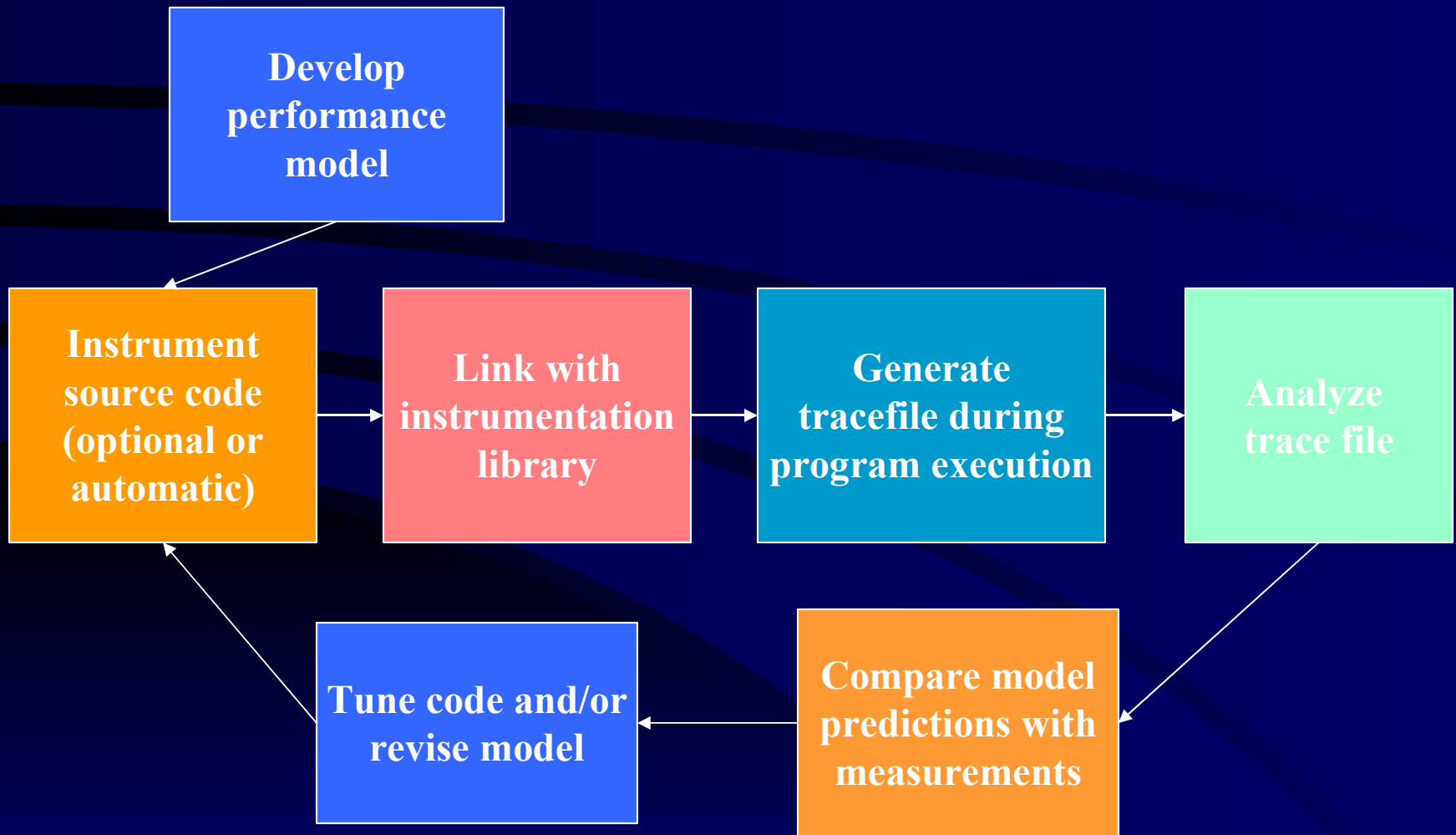
gprof

- Compile your code with `-pg`
- After execution the CWD will contain `gmon.out.[x]`
- Type `gprof`, it will look for `gmon.out` in the CWD. Otherwise give it name(s) with the `-m option`

prof/gprof

- All procedures called by the object code, many will be foreign to the programmer.
- Statistics are created by sampling and then looking up the PC and correlating it with the address space information.
- Phase problems may cause erroneous results and reporting.

Tool-assisted Performance Analysis



Parallel Performance Tool Capabilities

	<i>Post-mortem analysis</i>	<i>Automatic run-time analysis</i>	<i>Source code clickback</i>	<i>State-time diagram (zooming/scrolling)</i>	<i>Statistical analysis</i>	<i>Languages & platforms supported</i>
<i>AIMS</i>	X		X	X	X	Fortran, C SGI , Sun, IBM SP
<i>nupshot</i>	X			X(X)		Language-independent Most MPI platforms
<i>Paradyn</i> <i>(MPI version)</i>		X			X	Fortran, C, HPF, C++ IBM SP
<i>SvPablo</i>	X		X		X	ANSI C, HPF Sun, SGI
<i>VAMPIR</i>	X			X(X)		Language-independent all platforms

Speedshop

- `ssusage` collects information about your program's use of machine resources.
- `ssrun` allows you to run experiments on a program to collect performance data.
- `prof` analyzes the performance data you have recorded using `ssrun` and provides formatted reports.

Speedshop (cont)

- pixie instruments an executable to enable basic block counting experiments to be performed.
- squeeze allocates a region of virtual memory and locks the virtual memory down into real memory, making it unavailable to other processes.
- thrash allows you to explore paging behavior by allowing you to allocate a block of memory, then accessing the allocated memory to explore paging behavior.

Using Speedshop

- 1 Build the Application
- 2 Run Experiments on the application to collect performance Data
- 3 Examine the Performance Data
- 4 Generate an improved version of the program
- 5 Repeat as needed

Pcsamp Example

Profile listing generated Fri Jan 30 02:06:07 1998
with: prof nn0.pcsamp.21081

samples time CPU FPU Clock N-cpu S-interval Countsize
1270 13s R10000 R10010 195.0MHz 1 10.0ms 2(bytes)
Each sample covers 4 bytes for every 10.0ms (0.08% of 12.7000s)

-p[rocedures] using pc-sampling.

Sorted in descending order by the number of samples in each procedure.

Unexecuted procedures are excluded.

samples	time(%)	cum time(%)	procedure (dso:file)
1268	13s(99.8)	13s(99.8)	main (nn0:nn0.c)
1	0.01s(0.1)	13s(99.9)	_doprnt (/usr/lib32/libc.so.1:doprnt.c)

Example of Ustime

Profile listing generated Fri Jan 30 02:11:45 1998
with: prof nn0.ustime.21077

Total Time (secs) : 3.81
Total Samples : 127
Stack backtrace failed: 0
Sample interval (ms) : 30
CPU : R10000
FPU : R10010
Clock : 195.0MHz
Number of CPUs : 1

index	%Samples	self	descendants	total	name
[1]	100.0%	3.78	0.03	127	main
[2]	0.8%	0.00	0.03	1	__gettimeofday
[3]	0.8%	0.03	0.00	1	BSD getime

Speedshop

- `ssusage` collects information about your program's use of machine resources.
- `ssrun` allows you to run experiments on a program to collect performance data.
- `prof` analyzes the performance data you have recorded using `ssrun` and provides formatted reports.

Speedshop (cont)

- pixie instruments an executable to enable basic block counting experiments to be performed.
- squeeze allocates a region of virtual memory and locks the virtual memory down into real memory, making it unavailable to other processes.
- thrash allows you to explore paging behavior by allowing you to allocate a block of memory, then accessing the allocated memory to explore paging behavior.

Using Speedshop

- 1 Build the Application
- 2 Run Experiments on the application to collect performance Data
- 3 Examine the Performance Data
- 4 Generate an improved version of the program
- 5 Repeat as needed

Ideal experiment

Prof run at: Fri Jan 30 01:59:32 1998

Command line: prof nn0.ideal.21088

3954782081: Total number of cycles

20.28093s: Total execution time

2730104514: Total number of instructions executed

1.449: Ratio of cycles / instruction

195: Clock rate in MHz

R10000: Target processor modeled

Procedures sorted in descending order of cycles executed.
Unexecuted procedures are not listed. Procedures
beginning with *DF* are dummy functions and represent
init, fini and stub sections.

cycles(%)	cum %	secs	instrns	calls	procedure(dso:file)
3951360680(99.91)	99.91	20.26	2726084981	1	main(nn0.pixie:nn0.c)
1617034(0.04)	99.95	0.01	1850963	5001	doprnt (./libc.so.1.pixn32:doprnt.c)

Pcsamp Example

Profile listing generated Fri Jan 30 02:06:07 1998

with: prof nn0.pcsamp.21081

samples	time	CPU	FPU	Clock	N-cpu	S-interval	Countsize
1270	13s	R10000	R10010	195.0MHz	1	10.0ms	2(bytes)

Each sample covers 4 bytes for every 10.0ms (0.08% of 12.7000s)

-p[rocedures] using pc-sampling.

Sorted in descending order by the number of samples in each procedure.

Unexecuted procedures are excluded.

samples	time(%)	cum time(%)	procedure (dso:file)
1268	13s(99.8)	13s(99.8)	main (nn0:nn0.c)
1	0.01s(0.1)	13s(99.9)	__doprnt (/usr/lib32/libc.so.1:doprnt.c)

Example of Uertime

Profile listing generated Fri Jan 30 02:11:45 1998

with: prof nn0.uptime.21077

Total Time (secs) : 3.81
Total Samples : 127
Stack backtrace failed: 0
Sample interval (ms) : 30
CPU : R10000
FPU : R10010
Clock : 195.0MHz
Number of CPUs : 1

index	%Samples	self	descendents	total	name
[1]	100.0%	3.78	0.03	127	main
[2]	0.8%	0.00	0.03	1	_gettimeofday
[3]	0.8%	0.03	0.00	1	BSD_gettime

tprof for the SP2

- Reports CPU usage for programs and system. i.e.
 - All other processes while your program was executing
 - Each subroutine of the program
 - Kernel and Idle time
 - Each line of the program
- We are interested in source statement profiling.

tprof for the SP2

- Also based on sampling, which may cause erroneous reports.
- Compile using `-qlist` and `-g`.
- `tprof <program> <args>`
- Leaves a number of files in the CWD preceded by

___.

___h.<file>.c - Hot line profile

___t.<subroutine>_<file>.c - Subroutine profile

___t.main_<file>.c - Executable profile

PAT for the T3E

- Performance analysis tool is a low-overhead method for
 - Estimating time in functions
 - Determining load balance
 - Generating and viewing trace files
 - Timing individual calls
 - Displaying hardware performance counter information

PAT for the T3E

- Uses the UNICOS/mk `profil()` system call to gather information by periodically sampling and examining the program counter.
- Works on C, C++ and Fortran executables
- No recompiling necessary
- Just link with `-lpat`

Apprentice for the T3E

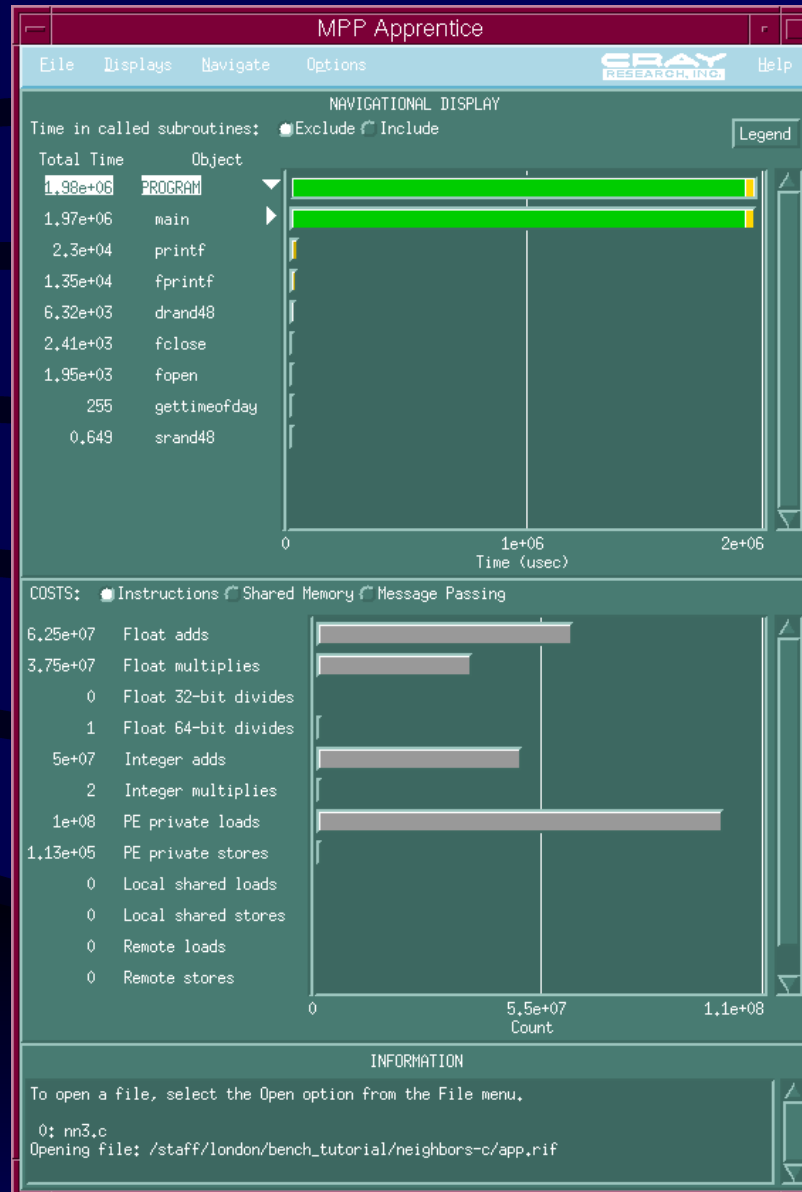
- Graphical interface for identifying bottlenecks.

```
% f90 -eA <file>.f -lapp
```

```
% cc -happrentice <file>.c  
-lapp
```

```
% a.out
```

```
% apprentice app.rif
```

Automated Instrumentation and Monitoring System (AIMS)

URL	http://science.nas.nasa.gov/Software/AIMS
Version	3.7
Languages	ANSI C, Fortran 77
Platforms	IBM SP with IBM MPI or MPICH Sun, SGI, and HP workstations with MPICH SGI Power Challenge with SGI MPI

AIMS Components

- Source code instrumentors
 - *xinstrument*
 - *batch_inst*
- Monitoring library
- Analysis tools
 - View Kernel (*VK*)
 - *tally* statistics generator

xinstrument

- GUI allows user to select specific source code constructs to be instrumented
- Default is to instrument all communication routines
- Other possibilities
 - All subroutines
 - All I/O
 - Enable by Type
 - Point and click on particular constructs in Construct Tree diagrams

xinstrument (cont.)

- Regards source code as nested collection of constructs
 - conditionals
 - loops
 - subroutines
 - communication calls
- Instrumented construct is replaced or surrounded by calls to AIMS monitor routines
- Execution of instrumented construct generates time-stamped event

Visualizing Trace Files with VK

- View Kernel (VK) animates a trace file
- VCR-like controls for tracefile playback
- Can set breakpoints by time or in specific source code constructs
- Source code click-back capability
- Timeline display
- Spokes view animates messages passed between tasks

Controlling Scale and Speed of Playback

- No scrolling or zooming capabilities
- Set *jump factor* between 0 and 1 to speed up animation
- Set pause times or breakpoints to slow down animation
- Set scale to view larger or smaller time interval (default is 100 milliseconds)

tally [options] [sorted tracefile]

where options include:

- | | |
|--------------|--|
| -help | Prints usage message |
| -proc[=Name] | Print information for procedure(s) |
| -node[=Node] | Print information for node(s) |
| -ncpu | Print information about normalized cpu usage |
| -msg | Print information about message sizes per node |
| -all | Print all information (proc+node+ncpu+msg) |

tally output - tally.summary

- Information for each procedure/function:
 - busy time: time spent performing useful work
 - global blocking: time spent in global blocking operation
 - send blocking: time spent in send operation
 - receive blocking: time spent in receive operation
 - life time: exclusive time
 - percentage communication: percentage of total execution time spent in communication
 - communication index: time spent in routine with respect to total time of program, as well as percentage of time spent in communication in this routine

tally.summary (cont.)

- Information for each node (and routine):
 - busy time
 - global blocking
 - send blocking
 - recv blocking
 - percentage communication

tally output - ncpu.summary

- NCPU for a given subroutine and a given k is the amount of CPU time used by that subroutine when k processors are busy, divided by k.
- Routine Concurrency - amount of time spent by each subroutine when k copies were executing simultaneously (indicates degree to which each routine was parallelized)

MPE Logging/nupshot

URL	http://www.mcs.anl.gov/mpi/mpich/
Version	1.1, April 1997
Languages	Language-independent
Tested platforms	SGI PCA and Origin 2000 IBM SP Sun Solaris

MPE Logging/nupshot

- Included with MPICH 1.1 distribution
- Distributed separately from rest of MPICH from PTLIB
- MPE logging library produces trace files in ALOG format
- nupshot display trace files in ALOG or PICL format
- Minimal documentation in MPICH User's Guide and man pages

MPE Logging Library (cont.)

- MPI application linked with liblmpi.a produces trace file in ALOG format
 - Calls to `MPE_Log_event` store event records in per-process memory buffer
 - Memory buffers are collected and merged during `MPI_Finalize`
- `MPI_Pcontrol` can be used to suspend and restart logging

nupshot

- Current version requires Tcl 7.3 and Tk 3.6
- Must be built with -32 on SGI IRIX
- Visualization displays
 - Timeline
 - Mountain Ranges
 - State duration histograms
- Zooming and scrolling capabilities

Pablo Project

- <http://www-pablo.cs.uiuc.edu/Projects/Pablo/>
- Goal: portable performance data environment for parallel systems
- Pablo Version 5.0 components
 - SDDF Library
 - TraceLibrary
 - I/O Analysis programs
 - Analysis GUI
 - SvPablo

Pablo TraceLibrary

- Extensions provide wrapper functions for management of event ID's for various event types
- Procedure and loop tracing done manually by inserting calls to TraceLibrary routines into application source code
- Default mode is to dump trace buffer contents to a trace file, but it's possible to have trace data output sent to a socket for real-time analysis

I/O Extension to TraceLibrary

- I/O instrumentation requires changes to application source code
- I/O trace initialization and termination routines must be called before and after calling any other I/O trace routines
- I/O trace bracketing routines provided for I/O requests that are not implemented as library calls (e.g., `getc` macro in C and Fortran I/O statements that are part of the language)

I/O Extension (cont.)

- I/O instrumentation options for C programs
 - Manually replace standard I/O calls with tracing counterparts
 - Define IOTRACE so that pre-processor replaces standard I/O calls with tracing counterparts
- I/O instrumentation of Fortran programs
 - Manually bracket each I/O call with I/O trace library bracketing routines

I/O Extension (cont.)

- Programs containing to I/O extension interface routines must be linked with
 - Pablo Trace Extension Library
libPabloTraceExt.a
 - Pablo Base Trace Library **libPabloTrace.a**

MPI TraceLibrary Extension

- MPI profiling library that can be linked in without making source code changes
- Each MPI process output a trace file labeled with the process number
- Insert call to **SetTraceFileName()** immediately after `MPI_Init()` to control location of trace file

MPI Extension (cont.)

- Disable tracing by calling `MPI_Control(0)`
- Re-enable tracing by calling `MPI_Control(1)`
- Link with Pablo Trace Extension Library (**`libPabloTraceExt.a`**) and Pablo Base Trace Library (**`libPabloTrace.a`**)
- Merge per-process trace file using the SDDF utility **`MergePabloTraces`**

Pablo Trace File Analysis

- Command-line FileStats program scans SDDF file and reports record types, min and max values for each field, and count of each record type.
- SDDFStatistics GUI for generating and browsing statistics from an SDDF file
- Pablo I/O analysis command-line routines
- Pablo Analysis GUI

SDDFStatistics

- Statistics for entire file are displayed along top of display
- Record types are displayed in panel at lower left
- Clicking on a record type brings up statistics for each field of that record type
- Clicking on a field displays a histogram summarizing values for that field
- Clicking on an array field type brings up statistics for each dimension of that field

SDDFStatistics Usage

- SDDFStatistics [-toolkitoption ...] [-loadSummary filename] [-openSDDF filename]
- Or use runSDDFStatistics script which invokes the SDDFStatistics program after setting environment variables so that required resources can be located

I/O Analysis Programs

- **Iostats** generates a report of application I/O activity summarized by I/O request type.
- **IOstatsTable** produces table summarizing information about I/O operations.
- **IOtotalsByPE** produces a report showing the total count, duration, and bytes involved for various operations by processor.

I/O Analysis Programs (cont.)

- **LifetimeIOstats** produces a report summarizing I/O activity by processor and file, prints a histogram of the file lifetimes, and prints total time spent in I/O calls for each procedure.
- **FileRegionIOstats** generates a report of application I/O activity summarized by file region. Each file is divided spatially into *regions* whose size is set by calling **enableFileRegionSummaries** (0).

I/O Analysis Programs (cont.)

- **TimeWindowIOstats** produces a report from Time Window Summary trace records. The execution time of the program is divided into *time windows* whose size is set by calling **enableTimeWindowSummaries()**.
- **SyncIOfileIDs** processes a trace file containing I/O trace events where many different file IDs may be associated with a given file, and write a new file where every I/O trace event associated with a particular file (as determined by the file name) has the same file ID.

Pablo Analysis GUI

- Toolkit of data transformation modules capable of processing SDDF records
- Supports graphical connection of performance data transformation modules in style of AVS
- By graphically connecting modules and interactively selecting trace data records, user specifies desired data transformation and presentations
- Expert users can develop and add new data analysis modules

Analysis GUI (cont.)

- Module types
 - Data analysis
 - Mathematical transforms (counts, sums, ratios, max, min, average, trig functions, etc.)
 - Synthesis of vectors and arrays from scalar input data
 - Data presentation - bar graphs, bubble charts, strip charts, contour plots, interval plots, kiviatic diagrams, 2-d and 3-d scatter plots, matrix displays, pie charts, polar plots

Paradyn

URL	http://www.cs.wisc.edu/paradyn/
Version	Release 2.0, September 1997
Languages	Fortran, Fortran 90, HPF, C, C++
Platforms	Sun SPARC (PVM version only) Windows NT on x86 IBM RS6000 and SP with AIX 4.1 or greater

Paradyn Goals

- Performance measurement tool that
 - scales to long-running programs on large parallel and distributed systems
 - automates much of the search for performance bottlenecks
 - avoids space and time overhead of trace-based tools

Paradyn Approach

- Dynamically instrument application
- Automatically control instrumentation in search of performance problems
- Look for high level problems (e.g., too much synchronization blocking, I/O blocking, or memory delays) using small amount of instrumentation
- Once general problem is found, selectively insert more instrumentation to find specific causes

Paradyn Components

- Front end and user interface that allow user to
 - display performance visualization
 - use the Performance Consultant to find bottlenecks
 - start and stop the application
 - monitor status of the application
- Paradyn daemons
 - monitor and instrument application processes
 - pvmd, mpid, winntd

Performance Consultant

- Based on W3 Search Model
 - “Why” - type of performance problems
 - “Where” - where in the program these problems occur
 - “When” - time during execution during which problems occur

Performance Consultant (cont.)

- Automatically locates potential bottlenecks in your application
 - Contains definitions of a set of performance problems in terms of *hypotheses* - e.g., PerfMetricX > Specified Threshold
 - Continually selects and refines which performance metrics are enabled and for which foci
 - Reports bottlenecks that exist for significant portion of phase being measured

SvPablo

URL	http://www-pablo.cs.uiuc.edu/
Version	Pablo release 5.0
Languages	HPF, ANSI C
Platforms	SGI running IRIX 6 Sun SPARC running Solaris

SvPablo

- Source view Pablo
- GUI for instrumenting source code and viewing runtime performance data
- Joint work at Univ. of Illinois and Rice Univ.
- HPF programs automatically instrumented by PGI HPF compiler
- C programs interactively instrumented using GUI
- Fortran 77/90 parser developed at JPL

SvPablo Project

- Set of application source files
- One or more *performance contexts*
- Performance context includes
 - instrumentation specification - source code points where performance measurements are to be inserted
 - set of performance data files in SDDF, generated when instrumented code is run
- Source files and parser options are shared among all contexts.

Line Metrics

- Count
- Duration
- Exclusive Duration
- Message Send Duration
- Message Send Count
- Message Send Size
- Message Receive Duration
- Message Receive Count
- Message Receive Size

VAMPIR

URL	http://www.pallas.de/pages/vampir.htm
Version	VAMPIR 1.0, VAMPIRtrace 1.5
Languages	Language-independent
Platforms	All major HPC platforms

VAMPIR

- Visualization and Analysis of MPI Resources
- Commercial tool from PALLAS GmbH
- VAMPIRtrace - MPI profiling library
- VAMPIR - trace visualization tool

VAMPIR Displays

- Process State Display
- Statistics Display
- Timeline Display
- Communications Statistics
- Configured by using
 - Pull-down menus
 - Configuration file

References

<http://www.nersc.gov>

<http://www.mhpcc.gov>

<http://www-jics.cs.utk.edu>

<http://www.tc.cornell.edu>

<http://www.netlib.org>

<http://www.ncsa.uiuc.edu>

<http://www.cray.com>

<http://www.psc.edu>

Additional Documentation

<http://www.cs.utk.edu/~mucci/MPPopt.html>

References

Hennessey and Patterson: *Computer Architecture, A Quantitative Approach*

Dongarra et al: MPI, *The Complete Reference*

Dongarra et al: PVM, *Parallel Virtual Machine*

Vipin Kumer et al: *Introduction to Parallel Computing*