Performance Optimization for the Origin 2000

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

Philip Mucci (mucci@cs.utk.edu)
Kevin London (london@cs.utk.edu)

University of Tennessee, Knoxville

Army Research Laboratory, Aug. 31 - Sep. 2

Outline

- Introduction to Performance Optimization
- Origin Architecture
- Performance Metrics and Issues
- Compiler Technology
- Numerical Libraries
- Performance Tools

Performance

- What is performance?
  - Latency
  - Bandwidth
  - Efficiency
  - Scalability
  - Execution time
- At what cost?

Performance Examples

- Operation Weather Forecasting Model
  - Scalability
- Database search engine
  - Latency
- Image processing system
  - Throughput

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

- Optimize compiler switches
- Integrate libraries
- Profile
- Optimize blocks of code that dominate execution time
- Always examine correctness at every stage!

Performance Strategies

- Always use optimal or near optimal algorithms.
  - Be careful of resource requirements and problem sizes.
- Maintain realistic and consistent input data sets/sizes during optimization.
- Know when to stop.

The 80/20 Rule

- Program spends 80 % time in 20 % of its code
- Programmer spends 20 % effort to get 80 % of the total speedup possible in the code.

How high is up?

- Profiling reveals percentages of time spent in CPU and I/O bound functions.
- Correlation with representative low-level, kernel and application benchmarks.
- Literature search.
- Peak speed of CPU means little in relation to most codes.
- Example: ISIS solver package

Don’t Sweat the Small Stuff

- Make the Common Case Fast (Hennessy)

PROCEDURE	TIME
main()	13%
procedure1()	17%
procedure2()	20%
procedure3()	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

- Machine configuration, libraries and tools
- Hardware and software overheads
- Alternate algorithms
- CPU/Resource requirements
- Amdahl's Law
- Communication pattern, load balance and granularity

Origin 2000 Architecture

- Up to 64 nodes
- Each node has 2 R10000's running at 195 Mhz, 1 memory per node
- Each R10000 has on chip 32K instruction, 32K data caches (32/64 Byte line)
- *Each* R10000 has a 4MB off-chip *unified* cache (128 Byte line)
- 64(58) entry TLB (each holds 2 pages)

Origin 2000 Architecture

- Each node is connected with a 624MB/sec *CrayLink*
- Shared memory support in hardware
- Variable page size, migration (*dplace)*
- Provides explicit (programmer) or implicit (compiler) parallelism
- Communication with MPI or shared-memory.

R10000 Architecture

- 5 independent, pipelined, execution units
- 1 non-blocking load/store unit
- 2 asymmetric integer units (both add, sub, log)
- 2 asymmetric floating point units (390 MFlops)
- Conditional load/store instructions

R10000 Architecture

- Superscalar with 5 pipelines
- Each pipeline has 7 stages
- Dynamic, out-of-order, speculative execution
- 32 logical registers
- 512 Entry Branch history table
- Hardware performance counters

Cache Architecture

- Small high-speed memories with block access
- Divided into smaller units of transfer called lines
- Address indicates
  - Page number
  - Cache line
  - Byte offset

Caches exploit Locality

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

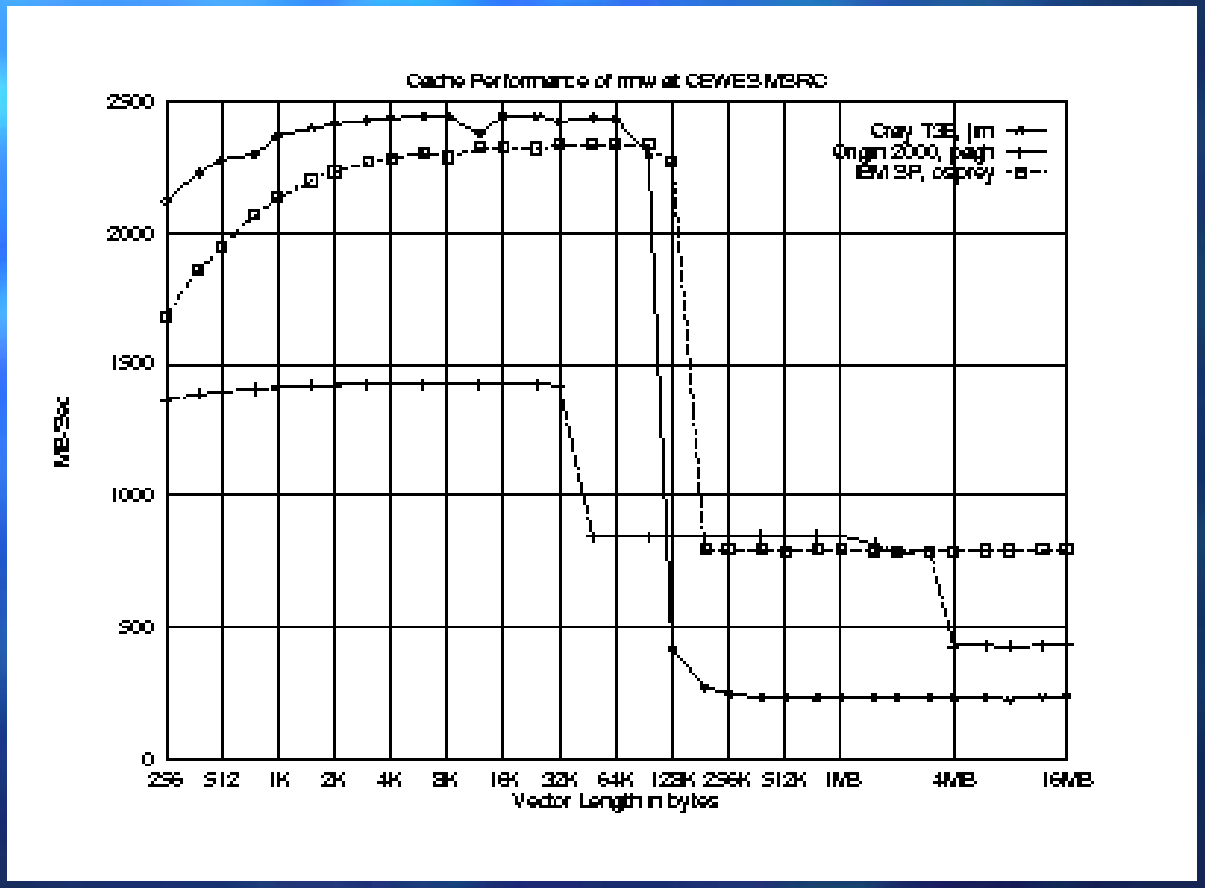
Temporal - If location X is being accessed, it is likely that X will be accessed again *soon*.

Cache Benchmark

http://www.cs.utk.edu/~mucci/cachebench

```
do i = 1,max_length
  start_time
  do j = 1,max_iterations
    do k = 1,i
      A(k) = i
    enddo
  enddo
  stop_time_and_print
enddo
```

Cache Performance



Cache Mapping

- Two major types of mapping
  - Direct Mapped

    Each memory address resides in only one cache line. (constant hit time)
  - N-way Set Associative

    Each memory address resides in one of N cache lines. (variable hit time)
  - Origin is 2-way set associative, 2-way interleaved

2 way Set Associative Cache

*distinct lines = size / line size * associativity*

Line0 Class0	Line1 Class0	Line2 Class0	Line3 Class0
Line0 Class1	Line1 Class1	Line2 Class1	Line3 Class1

Every datum can live in any class

but in only 1 line (computed from its address)

Which class? Least Recently Used

What is a TLB?

- Fully associative cache of virtual to physical address mappings. Used if data not in cache.
- Number is limited on R10K, by default:

16KB/pg * 2pgs/TLB * 58 TLBs = 2MB

- Processing with more than 2MB of data results in TLB misses.

O2K Memory Hierarchy

Origin 2000 Access Times

- Register: 1 cycle
- L1 Cache Hit: 2-3 cycles
- L1 Cache Miss: 7-13 cycles
- L2 Cache Miss: ~60-200 cycles
- TLB Miss: > 60 cycles

Performance Metrics

- **Wall Clock** time - Time from start to finish of our program
- MFLOPS - Millions of floating point operations per second
- MIPS - Millions of instructions per second
- Possibly ignore set-up cost

What about MFLOPS?

- Poor measures of comparison because
  - They are dependent on the definition, instruction set and the compiler
- Ok measures of numerical *kernel* performance for a single CPU

EXECUTION TIME

What do we use for evaluation

- For purposes of optimization, we are interested in:
  - Execution time of our code over a range of data sets
  - MFLOPS of our kernel code vs. peak in order to determine *EFFICIENCY*
  - Hardware resources dominating our 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...

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)

Basis for Performance Analysis

- Our evaluation will be based upon:
  - Performance of a single machine on a
  - Single (*optimal*) algorithm using
  - Execution time
- Optimizations are portable

Asymptotic Analysis

- Algorithm X requires O(N log N) time on O(N processors)
- This ignores constants and lower order terms!

10N > N log N for N < 1024

*10N*N < 1000N log N for N < 996*

Amdahl's Law

- The performance improvement is limited by the fraction of time the faster mode can be used.

Speedup = Perf. enhanced / Perf. standard

Speedup = Time sequential / Time parallel

Time parallel = T_{ser} + T_{par}

Amdahl's Law

- Be careful when using speedup as a metric. Ideally, use it only when the code is modified. Be sure to completely analyze and document your environment.
- Problem: This ignores the overhead of parallel reformulation.

Amdahl's Law

- Problem? This ignores scaling of the problem size with number of nodes.
- Ok, what about *Scaled Speedup?*
  - Results will vary given the nature of the algorithm
  - Requires O() analysis of communication and run-time operations.

Efficiency

- A measure of code quality?

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

$$S = P * E$$

- Sequential time is not a good reference point. For Origin, 4 is good.

Issues in Performance

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

Issues in Performance

- Cache utilization
- Register allocation
- Loop nest optimization
- Instruction scheduling and pipelining
- *Compiler Technology*
- Programming Model (Shared Memory, Message Passing)

Problem Size and Precision

- Necessity
- Density and Locality
- Memory, Communication and Disk I/O
- Numerical representation
  - INTEGER, REAL, REAL*8, REAL*16

Parallel Performance Issues

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

Understanding Compilers

- Why?
  - Compilers emphasize correctness rather than performance
  - On well recognized constructs, compilers will *usually* do better than the developer
  - The idea? To express an algorithm *clearly* to the compiler allows the most optimization.

Compiler Technology

- Ideally, compiler should do most of the work
- Rarely happens in practice for *real* applications
- Here we will cover some of the options for the MIPSpro 7.x compiler suite

Recommended Flags

```
-n32 -mips4 -Ofast=ip27 -LNO:cache_size2=4096
            -OPT:IEEE_arithmetic=3
```

- Use at link and compile time
- We don’t need more than 2GB of data
- Turn on the highest level of optimization for the Origin
- Tell compiler we have 4MB of L2 cache
- Favor speed over precise numerical rounding

Accuracy Considerations

- Try moving forward

```
-O2 -IPA -SWP:=ON
-LNO -TENV:X=0-5
```

- Try backing off

```
-Ofast=ip27
-OPT:roundoff=0-3
-OPT:IEEE_arithmetic=1-3
```

Compiler flags

- Many optimizations can be controlled separately from -Ofast
- It's better to selectively disable optimizations rather than reduce the level of global optimization
- -OPT:IEEE_arithmetic=n controls rounding and overflow
- -OPT:roundoff=n controls roundoff

Roundoff example

- Floating point arithmetic is not associative. Which order is correct?
- Think about the following example:

```
sum = 0.0
do i = 1, n
   sum = sum + a(i)
enddo
```

```
sum1 = 0.0
sum2 = 0.0
do i = 1, n-1, 2
   sum1 = sum1 + a(i)
   sum2 = sum2 + a(i+1)
enddo
sum = sum1 + sum2
```

Exceptions

- Numerical computations resulting in undefined results
- Exception is generated by the processor (with control)
- Handled in software by the Operating System.

Exception profiling

- If there are few exceptions, enable a faster level of exception handling at compile time with `-TENV:X=0-5`
- Defaults are `1` at `-O0` through `-O2`, `2` at `-O3` and higher
- Else if there are exceptions, link with `-lfpe`

```
setenv TRAP_FPE "UNDERFL=ZERO"
```

Aliasing

- The compiler needs to assume that any 2 pointers can point to the same region of memory
- This removes many optimization opportunities
- `-Ofast` implies `-OPT:alias=typed`
- Only pointers of the same type can point to the same region of memory.

Advanced Aliasing

- Programmer knows much more about pointer usage than compiler.
- `-OPT:alias=restrict` - all pointer variables are assumed to point to non-overlapping regions of memory.
- `-OPT:alias=disjoint` - all pointer expressions are assumed to point to non-overlapping regions of memory.
- Very important for C programs.

Advanced Aliasing

- Most advanced form is the `ivdep` compiler directive.
- Used on inner loops with software pipelining.
- Can move a loop to be completely load/store bound.
- Please refer to the *Origin 2000 Optimization and Tuning Guide*.

Software Pipelining

- Important contribution of `-O3`
- Different iterations of a loop are overlapped in time in an attempt to keep all the functional units busy.
- Data needs to be in cache for this to work well.
- Can be enabled with `-SWP:=ON`

Interprocedural Analysis

- When analysis is confined to a single procedure, the optimizer is forced to make worst case assumptions about the possible effects of subroutines.
- IPA analyzes the entire program at once and feeds that information into the other phases.

IPA features

- Inlining across source files
- Common block padding
- Constant propagation
- Dead function/variable elimination
- Library reference optimizations
- Enabled with `-IPA`

Inlining

- Replaces a subroutine call with the function itself.
- Useful in loops that have a large iteration count and functions that don’t do a lot of work.
- Allows other optimizations.
- Most compilers will do inlining but the decision process is conservative.

Manual Inlining

`-INLINE:file=<filename>`

`-INLINE:must=<name>[,name2,name3..]`

`-INLINE:all`

- Exposes internals of the call to the optimizer
- Eliminates overhead of the call
- Expands code

Loop Nest Optimizer

- Optimizes the use of the memory heirarchy
- Works on relatively small sections of code
- Enabled with `-LNO`
- Visualize the transformations with

  `-FLIST:=on`

  `-CLIST:=on`

LNO functionality

- Cache blocking
- Merging of data used together
- Loop fusion
- Loop unrolling
- Loop interchange
- Loop fission
- Prefetching

Optimized Arithmetic Libraries

- Advantages:
  - Subroutines are quick to code and understand.
  - Routines provide *portability.*
  - Routines perform well.
  - Comprehensive set of routines.
- Disadvantages
  - Can lead to vertical code structure
  - May mask memory performance problems

Numerical Libraries

- libfastm
  - Link with `-r10000` and `-lfastm`
  - Link before `-lm`
- CHALLENGEcomplib and SCSL
  - Sequential and parallel versions
  - FFTs, convolutions, BLAS, LINPACK, EISPACK, LAPACK and sparse solvers

CHALLENGEcomplib and SCSL

- Serial
  - `-lcomplib.sgimath` or
  - `-lscs`
- Parallel
  - `-mp -lcomplib.sgimath_mp` or
  - `-lscs_mp`

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`

ScaLAPACK

- Parallelized LAPACK routines
- Based upon LAPACK and BLAS
- Can be used with vendor libraries
- Available in

`/home/army/susan/ECKERT/lib`

PETSc

- Generalized sparse solver package for solution of PDEs.
- Multiple preconditioners and explicit and implicit methods.
- Available in

`/home/army/susan/ECKERT/petsc-2.0.21`

`http://www.mcs.anl.gov/petsc`

O2K Performance Tools

- Timers
- Hardware Counters
- Profilers
  - perfex
  - SpeedShop
  - prof
  - dprof
  - cvd

External 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
- Use `timex` on the SGI.
- Warning! The definition of CPU time is different on different machines.

External 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.

Internal Timers

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

Internal Timers

- `clock_gettime()`
- `MPI_Wtime()` returns elapsed wall clock time in seconds as a double.

Fortran

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

Hardware Performance Counters

- 2 32-bit registers that do the counting
- 32 different events (30 distinct, 14 each, 1 shared)
- OS accumulates counts into 64-bit quantities
- Both user and kernel modes can be measured
- Explicit counting or overflows

Some Hardware Counter Events

- Cycles, Instructions
- Loads, Stores, Misses
- Exceptions, Mispredictions
- Coherency
- Issued/Graduated
- Conditionals

Hardware Performance Counter Access

- At the source level with raw counter API or perfex API.
- At the application level with `perfex`
- At the function level with SpeedShop and `prof`.
- List all the events with `perfex -h`

Origin Counter API

- Very simple, easy to use.
  - `start_counter()`
  - `stop_counter()`
  - `read_counters()`
  - `print_counters()`
- Information available with

```
man start_counters
```

Perfex usage

- Used to gather statistics about the entire run of the program.
- From the command line:

`perfex [`*`options`*`] command [`*`args`*`]`

- At compile time, perfex library calls can start or stop collection.
- Link with `-lperfex`
- `man libperfex`

Perfex features

- Explicit counts (FP and Total)

```
perfex -e 15 -e 21 <exe>
```

- Multiplex over all counts

```
perfex -a <exe>
```

- Analytic output (for all)

```
perfex -a -y <exe>
```

- Exceptions (for Cycles & L1DC misses)

```
perfex -e 1 -e 25 -x <exe>
```

Speedshop

- Find out exactly where program is spending it's time
  - procedures
  - lines
- Uses 3 methods
  - Sampling
  - Counting
  - Tracing

Speedshop Components

- 4 parts
  - `ssrun` performs experiments and collects data
  - `ssusage` reports machine resources
  - `prof` processes the data and prepares reports
  - SpeedShop allows caliper points
- See man pages

Speedshop Usage

`ssrun [options] <exe>`

- output is placed in *./command.experiment.pid*
- Viewed with

`prof [options] <command.experiment.pid>`

Speedshop Sampling

- Basd upon
  - interval timers
  - instructions
  - cycles
  - i/d/s cache misses
  - TLB faults
  - FP instructions
  - any hardware counter

`ssrun` Option

```
-usertime (30ms)
-pcsamp   (10ms)
-gi_hwc
-cy_hwc
-ic_hwc
-isc_hwc
-dc_hwc
-dsc_hwc
-tlb_hwc
-gfp_hwc
-prof_hwc
```

SpeedShop Sampling

- All procedures called by the 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 and symbol table information.
- Phase problems may cause erroneous results and reporting.

Speedshop Counting

- Based upon basic block profiling
- Basic block is a section of code with one entry and one exit
- Executable is instrumented with `pixie`
- `pixie` adds a counter to every basic block

Ideal Experiment

- `ssrun -ideal`
- Calculates ideal time
  - no cache/TLB misses
  - minimum latencies for all operations
- Exact operation count with `-op`
  - floating point **operations (MADD is 2)**
  - integer operations

Prof Usage

- Normally just `prof <output file>`
- `-heavy` lists offending line numbers
- `-source` lists source code and disassembled machine code with specific instructions highlighted

ideal Experiment Example

```
 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
------------------------------------------------------------
.
.
.
------------------------------------------------------------
       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
```

pcsamp Experiment 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)
------------------------------------------------------------
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
```

usertime Experiment Example

```
------------------------------------------------------------
Profile listing generated Fri Jan 30 02:11:45 1998
    with:       prof nn0.usertime.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_getime
```

Gprof Usage

- `prof` doesn't give information about the call heirarchy
- Some function may be used everywhere but is only a problem in one specific instance.
- `prof -gprof` can be used only with Ideal and Usertime experiments

Gprof information

- In addition to the information from prof
  - Contributions from descendants
  - Distribution relative to callers
- To get gprof like information use

```
prof -gprof <output file>
```

Exception Profiling

- By default the R10000 causes hardware traps on floating point exceptions and then ignores them in software
- This can result in lots of overhead.
- Use `ssrun -fpe <exe>` to generate a trace of locations generating exceptions.

Address Space Profiling

- Used primarily for checking shared memory programs for memory contention.
- Generates a trace of most frequently referenced pages
- Samples operand address instead of PC

```
dprof -hwpc <exe>
```

dprof

- Output is organized by
  - virtual address
  - thread
  - samples per page
- Difficult to trace pages to actual symbols

Parallel Profiling

- After tuning for a single CPU, tune for parallel.
- Use full path of tool
- `ssrun/perfex` used directly with `mpirun`

- `mpirun <opts> /bin/perfex -mp <opts> <exe> <args> |& cat > output`
- `mpirun <opts> /bin/ssrun <opts> <exe> <args>`

Parallel Profiling

- `perfex` outputs all tasks followed by all tasks summed
- In shared memory executables, watch
  - load imbalance (cntr 21, flinstr)
  - excessive synchronization (4, store cond)
  - false sharing (31, shared cache block)

CASEVision Debugger

- cvd
- GUI interface to SpeedShop PC sampling and ideal experiments
- Interface to viewing automatic parallelization options
- Poor documentation
- Debugging support
- This tool is complex...

Outline

- Performance guidelines
- Array/Loop Optimization
- Language specific considerations
- MPI Optimization
- Shared Memory Optimization

Guidelines for Performance

- I/O is slow
- System calls are slow
- Use your in-cache data completely
- When looping, remember the pipeline!
  - Branches
  - Function calls
  - Speculation/Out-of-order execution
  - Dependencies

Array Optimization

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

Memory Access

- Programs should be designed for maximal cache benefit.
  - Stride 1 access patterns
  - Use entire cache lines
  - Reusing data as soon as possible after first reference
- Also, we should minimize page faults and TLB misses. (code and `dplace`)

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 Referencing

- In C, outer most index should change fastest.

[x,**Y**]

- In Fortran, inner most index should change fastest.

(**X**,y)

Array Initialization

Which to choose?

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

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 Padding

- Data in `COMMON` blocks is allocated contiguously
- Watch for powers of two and know the associativity of your cache
- Example: dot product, possible miss per element on 16KB Direct mapped cache for 4 byte elements

```
common /xyz/ a(2048),b(2048)
```

Array Padding

*a = a + b * c*

	Tuned	Untuned	Tuned -O3	Untuned -O3
Origin 2000	1064.1	1094.7	800.9	900.3

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, *innermost index changes fastest*.

Stride Minimization

- Stride 1

```
do y = 1, 1000
   do x = 1, 1000
        c(x,y) = c(x,y) + a(x,y)*b(x,y)
```

- Stride 1000

```
do y = 1, 1000
   do x = 1, 1000
        c(y,x) = c(y,x) + a(y,x)*b(y,x)
```

Stride Minimization

	Untuned -O3	Tuned -O3
Origin 2000	67.24	23.27
IBM SP2	201.07	17.54
Cray T3E	37.61	37.66

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

```
do i = 1, 50000
  x = x * a(i) + b(i)
enddo
do i = 1, 100000
  y = y + a(i) / b(i)
enddo
```

- Tuned

```
do i = 1, 50000
   x = x * a(i) + b(i)
   y = y + a(i) / b(i)
enddo
do i = 50001, 100000
  y = y + a(i) / b(i)
enddo
```

Loop Fusion

	Untuned -O3	Tuned -O3
Origin 2000	276.37	191.06
IBM SP2	254.96	202.76
Cray T3E	1405.52	1145.91

Loop Interchange

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

Loop Interchange

- Untuned

```
real*8 a(2,40,2000)

do i=1, 2000
 do j=1, 40
   do k=1, 2
     a(k,j,i) = a(k,j,i)*1.01
   enddo
 enddo
enddo
```

- Tuned

```
real*8 a(2000,40,2)

do i=1, 2
 do j=1, 40
   do k=1, 2000
     a(k,j,i) = a(k,j,i)*1.01
   enddo
 enddo
enddo
```

Loop Interchange

	Untuned -O3	Tuned -O3
Origin 2000	73.85	55.23
IBM SP2	432.39	434.15
Cray T3E	241.85	241.80

Floating IF's

- IF statements that do not change from iteration to iteration may be moved out of the loop.
- Compilers can usually do this except when
  - Loops contain calls to procedures
  - Variable bounded loops
  - Complex loops

Floating IF's

- Untuned

```
do i = 1, lda
  do j = 1, lda
    if (a(i) .GT. 100) then
      b(i) = a(i) - 3.7
    endif
      x = x + a(j) + b(i)
  enddo
enddo
```

- Tuned

```
do i = 1, lda
  if (a(i) .GT. 100) then
    b(i) = a(i) - 3.7
  endif
  do j = 1, lda
     x = x + a(j) + b(i)
  enddo
enddo
```

Floating IF's

	Untuned –O3	Tuned –O3
Origin 2000	203.18	94.11
IBM SP2	80.56	80.77
Cray T3E	160.86	161.21

Loop Defactorization

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

Gather-Scatter Optimization

- Untuned

```
do i = 1, n
  if (t(I).gt.0.0) then
    a(I)=2.0*b(I-1)
  end if
enddo
```

- Tuned

```
inc = 0
do i = 1, n
  tmp(inc) = i
  if (t(I).gt.0.0) then
    inc = inc + 1
  end if
enddo
do I = 1, inc
  a(tmp(I))=2.0*b((tmp(I)-1)
enddo
```

Gather-Scatter Optimization

- For loops with branches inside loops
- Increases pipelining
- Often, body of the loop is executed on every iteration, thus no savings
- Solution is to split the loop with a temporary array containing indices of elements to be computed with

IF Statements in Loops

- Solution is to unroll the loop
- Move conditional elements into scalars
- Test scalars at the end of the loop body

```
do I = 1, n, 2
  a = t(I)
  b = t(I+1)
  if (a .eq. 0.0)
  end if
  if (b .eq. 0.0)
  end if
end do
```

Loop Defactorization

- Note that floating point operations are not always associative.

$$(A + B) + C \quad != A + (B + C)$$

- Be aware of your precision
- Always verify your results with unoptimized code first!

Loop Defactorization

- Untuned

```
do i = 1, lda
   A(i) = 0.0
   do j = 1, lda
     A(i)=A(i)+B(j)*D(j)*C(i)
   enddo
enddo
```

- Tuned

```
do i = 1, lda
  A(i) = 0.0
  do j = 1, lda
    A(i) = A(i) + B(j) * D(j)
  enddo
  A(i) = A(i) * C(i)
enddo
```

Loop Defactorization

	Tuned -O3	Untuned -O3
Origin 2000	371.95	559.17
IBM SP2	449.03	591.26
Cray T3E	3201.35	3401.61

Loop Peeling

- For loops which access previous elements in arrays.
- Compiler often cannot determine that an item doesn't need to be loaded every iteration.

Loop Peeling

- Untuned

```
jwrap = lda
do i = 1, lda
  b(i) = (a(i)+a(jwrap))*0.5
  jwrap = i
enddo
```

- Tuned

```
b(1) = (a(1)+a(lda))*0.5
do i = 2, lda
   b(i) = (a(i)+a(i-1))*0.5
enddo
```

Loop Peeling

	Tuned -O3	Untuned -O3
Origin 2000	61.06	63.33
IBM SP2	25.68	40.50
Cray T3E	72.93	90.05

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

```
do i = 1, lda
   do j = 1, ldb
      do k = 1, ldc
         A(k,j,i) = A(k,j,i) + B(k,j,i) * C(k,j,i)
      enddo
   enddo
enddo
```

Loop Collapse

- Tuned

```
do i = 1, lda*ldb*ldc
   A(i,1,1) = A(i,1,1) + B(i,1,1) * C(i,1,1)
enddo
```

- More Tuned (declarations are 1D)

```
do i = 1, lda*ldb*ldc
   A(i) = A(i) + B(i) * C(i)
enddo
```

Loop Collapse

	Tuned	Tuned –O3	Tuned 2nd	Tuned 2nd –O3
Origin 2000	400.25	143.01	410.58	77.86
IBM SP2	144.75	31.57	144.18	31.54
Cray T3E	394.19	231.44	394.92	229.86

Loop Unrolling

- Data dependence delays can be reduced or eliminated.
- Reduce loop overhead.
- Usually performed well by the compiler or preprocessor.

Loop Unrolling

- Untuned

```
do i = 1, lda
   do j = 1, lda
      do k = 1, 4
         a(j,i) = a(j,i) + b(i,k) * c(j,k)
      enddo
   enddo
enddo
```

Loop Unrolling

- Tuned (4)

```
do i = 1, lda
   do j = 1, lda
      a(j,i) = a(j,i) +  b(i,1) * c(j,1)
      a(j,i) = a(j,i) +  b(i,2) * c(j,2)
      a(j,i) = a(j,i) +  b(i,3) * c(j,3)
      a(j,i) = a(j,i) +  b(i,4) * c(j,4)
   enddo
enddo
```

Loop Unrolling

	Tuned -O3	Untuned -O3
Origin 2000	61.06	63.33
IBM SP2	11.26	12.65
Cray T3E	36.30	24.41

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

```
do i = 1, lda
   do j = 1, lda
      a = a + (b(j) * c(i))
   enddo
enddo
```

Loop Unrolling and Sum Reductions

- Tuned (4)

```
do i = 1, lda
   do j = 1, lda, 4
      a1 = a1 + b(j) * c(i)
      a2 = a2 + b(j+1) * c(i)
      a3 = a3 + b(j+2) * c(i)
      a4 = a4 + b(j+3) * c(i)
   enddo
enddo
aa = a1 + a2 +a3 + a4
```

Loop Unrolling and Sum Reductions

	Untuned –O3	2 Tuned	2 Tuned –O3	4 Tuned -O3	8 Tuned -O3	16 Tuned -O3
Origin 2000	454	4945	352	350	350	**330**
IBM SP2	281	6490	563	281	281	**263**
Cray T3E	865	10064	564	340	**231**	860

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 flop requires two loads and one store.

```
do i = 1, lda
   do j = 1, ldb
      A(i,j) = B(i,j) * C(j)
   enddo
enddo
```

Outer Loop Unrolling

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

```
do i = 1, lda, 4
   do j = 1, ldb
      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)
   enddo
enddo
```

Outer Loop Unrolling

	Tuned -O3	Untuned -O3
Origin 2000	28.85	34.52
IBM SP2	74.67	286.11
Cray T3E	14.33	30.91

Cache Blocking

- Takes advantage of the cache by working with smaller tiles of data
- Only really beneficial on problems with significant potential for reuse
- Merges naturally with unrolling and sum-reduction

Cache Blocking

- Untuned

```
REAL*8 A(M,N)
REAL*8 B(N,P)
REAL*8 C(M,P)

DO J=1,P
 DO I=1,M
  DO K=1,N
    C(I,P) = C(I,P) +
    A(I,K)*B(K,J)
  ENDDO
 ENDDO
ENDDO
```

- Tuned

```
DO JB=1,P,16
 DO IB=1,M,16
  DO KB=1,N
   DO J=JB,MIN(P,JB+15)
    DO I=IB,MIN(M,IB+15)
      C(I,P) = C(I,P) +
      A(I,K)*B(K,J)
    ENDDO
   ENDDO
  ENDDO
 ENDDO
ENDDO
```

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 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

	Tuned -O3	Untuned -O3
Origin 2000	74.20	74.09
IBM SP2	40.69	74.71
Cray T3E	61.70	160.05

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

	Tuned -O3	Untuned -O3
Origin 2000	210.71	336.44
IBM SP2	422.65	494.05
Cray T3E	771.28	844.17

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

	Tuned -O3	Untuned -O3
Origin 2000	51.65	48.99
IBM SP2	57.43	57.40
Cray T3E	387.77	443.45

Subexpression Elimination
Parenthesis

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

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
Type Considerations

- Changes the type or precision of data.
  - Reduces resource requirements.
  - Avoid type conversions.
  - Processor specific performance.
- Do you really need 8 or 16 bytes of precision?

Subexpression Elimination
Type Considerations

- Consider which elements are used together?
  - Should you be merging your arrays?
  - Should you be splitting your loops for better locality?
  - For C, are your structures packed tightly in terms of storage and reference pattern?

F90 Considerations

- WHERE statements
- ARRAY syntax
- ALLOCATE placement
- OO complication
  - Class dependencies
  - Code fragmentation
  - Operator overloading
  - Inlining

C/C++ Considerations

- Use C++ I/O operators
- Call by const ref
- OO complication
- Avoid unsigned conversions
- Use `inline`, `const` and `__restrict` keywords

dplace Usage

- Used to specify different page sizes and data placement
- For performance use:

```
dplace -data_pagesize 64k -stack_pagesize 64k <program>

mpirun -np <procs> /usr/sbin/dplace <args> <program>

mpirun -np <procs> /bin/ssrun <args> /usr/sbin/dplace
   <args> <program>
```

Parallel Optimization

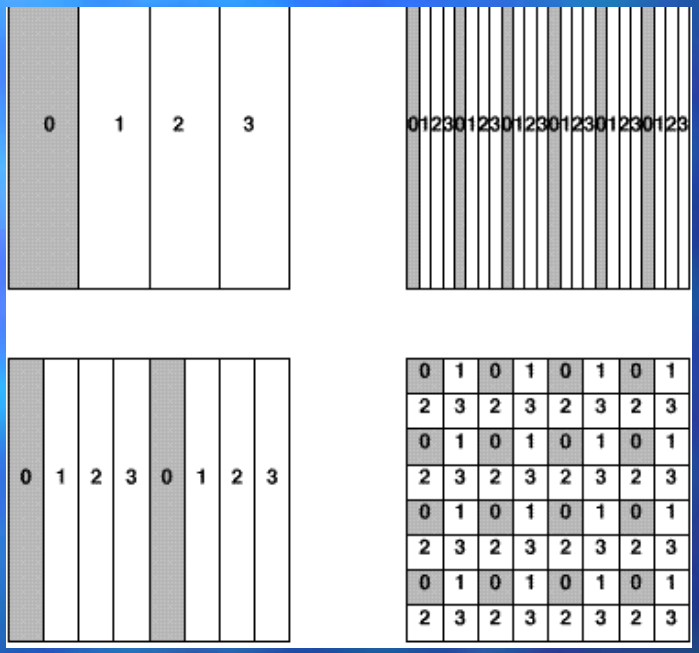
- Two programming models.
  - Message Passing
  - Shared Memory
- Optimizing parallel code

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 APIs

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

Message Passing APIs

- 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 Interface

- 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.
- 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.

Message Passing

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

MPI Message Passing

- MPI introduces communication **modes** dictating semantics of completion of send operations.
  - **B**uffered - When transmitted or buffered, space provided/limited by application, else error.
  - **R**eady - Only if receive is posted, else error.
  - **S**ynchronous - 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.
- Avoid one sided and persistent 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_SCHEDTYPE <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`

Additional Material

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

- Slides
- Optimization Guides
- Papers
- Pointers
- Compiler Benchmarks

HTTP 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

http://techpubs.sgi.com

References

SGI: *Origin 2000 Optimization and Tuning Guide*

SGI: *MIPSpro Compiler Performance Tuning Guide*

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*

IBM: *Optimization and Tuning Guide for Fortran, C, C++*