PGI and Cray Compiler Optimization

Go to Menu

Outline

Foreword

- Source code that is readable, maintainable, and produces correct results is not always organized for efficient execution. Normally, the first step in the program development process involves producing code that executes and produces the correct results. This first step usually involves compiling without much worry about optimization. After code is compiled and debugged, code optimization and parallelization become an issue.

- Invoking one of the PGI/GNU/Intel/Cray/Pathscale compiler commands with certain options instructs the compiler to generate optimized code. Optimization is not always performed since it increases compilation time and may make debugging difficult. However, optimization produces more efficient code that usually runs significantly faster than code that is not optimized.

Go to Menu

Outline: Introduction

- Parallel Compiling on Jaguar
- System Parallel Compilers
- Wrappers and Compiling Tips
- System Serial Compilers
- Default Compilers
- MPI Codes

Parallel Compiling on Jaguar

- Jaguar has two kinds of nodes:
  - Compute Nodes running the CNL OS
  - Service and login nodes running Linux
- To build a code for the compute nodes, you should use the Cray wrappers `cc`, `CC`, and `ftn`. The wrappers will call the appropriate compiler which will use the appropriate header files and link against the appropriate libraries. Use of wrappers is crucial for building the parallel codes on Cray.
- We highly recommend that the `cc`, `CC`, and `ftn` wrappers be used when building for the compute nodes! Both parallel and serial codes.
- To build a code for the Linux service nodes, you should call the compilers directly.
- We strongly suggest that you don't call the compilers directly if you are building code to run on the compute nodes.
- No long serial jobs should be run on service nodes, use compute nodes instead.

Go to Menu

System Parallel Compilers

The following compilers should be used to build codes on Jaguar!
Use these compilers

Language	Compiler
C	`cc`
C++	`CC`
Fortran 77, 90 and 95	`ftn`

Note that cc, CC and ftn are actually the Cray XT Series wrappers for invoking the PGI, GNU, Intel or Pathscale compilers (discussed later…)

Go to Menu

Wrappers and Compiling Tips

- Why to use wrappers to build (compile and link) the code:
  - Automatically point to correct compiler based on modules loaded
  - Wrappers automatically find and include paths and libraries of loaded modules (e.g., mpi, libsci)
- Use same makefile for all compilers*
- Calling base compilers directly (e.g., `pgf90`) results in serial code that runs only on login nodes
  - Not what you want! Use wrapper instead and run on compute nodes
  - Discourteous to other users to do production work on login nodes

* Except compiler-specific flags

Go to Menu

System Serial Compilers

- Available compilers:
  - Portland Croup (PGI). Module name: PrgEnv-pgi
    - ✓ `pgcc`
    - ✓ `pgCC`
    - ✓ `pgf90/pgf95`
    - ✓ `pgf77`
  - GNU. Module name: PrgEnv-gnu
    - ✓ `gcc`
    - ✓ `g++`
    - ✓ `Gfortran`
  - Intel. Module name: PrgEnv-intel
    - ✓ `icc` (c/c++ codes)
    - ✓ `ifort`
  - Cray compilers. Module name: PrgEnv-cray
    - ✓ `craycc`
    - ✓ `crayCC`
    - ✓ `crayftn`
  - Pathscale. Module name: PrgEnv-pathscale
    - ✓ `pathcc`
    - ✓ `pathCC`
    - ✓ `path90/pathf95` (only available if gcc/4.2.1 or higher is loaded)

Note that the `man` pages for the system compilers will only give the most basic information, i.e.

```
%man cc
%man CC
%man ftn
```

The `man` pages with the specific compiler options can be accessed by using the names of the serial compilers on this slide:

```
%man pgcc
%man g++
%man crayftn
```

Go to Menu

Default Compilers

- Default compiler is PGI. The list of all packages is obtained by
  - `module avail PrgEnv`
- To use the Cray wrappers with other compilers the programming environment modules need to be swapped, i.e.
  - `module swap PrgEnv-pgi PrgEnv-gnu`
  - `module swap PrgEnv-pgi PrgEnv-cray`
- To just use the GNU/Cray compilers directly load the GNU/Cray module you want:
  - `module load PrgEnv-gnu/2.1.50HD`
  - `module load PrgEnv-cray/1.0.1`
- It is possible to use the GNU compiler versions directly without loading the Cray Programming Environments, but note that the Cray wrappers will probably not work as expected if you do that.

Go to Menu

MPI Codes

- *All system compilers (PGI, GNU, Intel, Cray, Pathscale) can handle MPI standard specification parallel codes through the use of compiler wrappers (cc, ftn, CC)*
- MPT – Cray's MPI library
  - Use latest MPT (3.1.x)
- Default settings are set based on the best performance on most codes.
  - Some codes may benefit from setting or adjusting the environment variable settings.
- More information is available on man pages "man mpi"

Go to Menu

Outline: PGI® compilers

- Portland Group (PGI)
- List of the Compiler Option Categories
- PGI Basic Compiler Usage
- Flags to support language dialects
- Specifying the target architecture
- Flags for debugging aids
- Useful Compiler Flags

Go to Menu

Portland Group (PGI)

- Cray provides the Portland Group (PGI) compilers as part of several programming environments on Jaguar.
- PGI compilers are loaded by default.

List of the Compiler Option Categories

Description of the following options is provided by PGI man pages:

- Overall Options
- Optimization Options (covered in this document)
- Debugging Options
- Preprocessor Options
- Assembler Options
- Linker Options
- Language Options
- Target-specific Options

Go to Menu

PGI Basic Compiler Usage

- A compiler driver interprets options and invokes pre-processors, compilers, assembler, linker, etc.
- Options precedence: if options conflict, last option on command line takes precedence
- Use -Minfo and -Mneginfo to see a listing of optimizations and transformations performed by the compiler
- Use -help to list all options or see details on how to use a given option, e.g. pgf90 -Mvect -help
- Use man pages for more details on options, e.g. “man pgf90”
- Use –v to see under the hood

Go to Menu

Flags to support language dialects

- Fortran
  - ftn
  - Suffixes .f, .F, .for, .fpp, .f90, .F90, .f95, .F95
  - -Mextend, -Mfixed, -Mfreeform
  - Type size –i2, -i4, -i8, -r4, -r8, etc.
  - -Mcray, -Mbyteswapio, -Mupcase, -Mnomain, -Mrecursive, etc.
- C/C++
  - cc, CC
  - Suffixes .c, .C, .cc, .cpp, .i
  - -B, -c89, -c9x, -Xa, -Xc, -Xs, -Xt
  - -Msignextend, -Mfcon, -Msingle, -Muchar, -Mgccbugs

Go to Menu

Specifying the target architecture

- `-tp target` - Specify the type of the target processor;
- The default in the absence of the -tp flag is to compile for the type of CPU on which the compiler is running.
- The targets are:
  - `-tp k8-64` - AMD Opteron or Athlon-64 in 64-bit mode.
  - `-tp amd64e` - AMD Opteron revision E or later, in 64-bit mode; includes SSE3 instructions
  - `-tp x64` - Single binary where each procedure is optimized for the AMD Opteron in 64-bit mode; the selection of which optimized copy to execute is made at run time depending on the machine executing the code.
  - `-tp k8-32,k7,p7,piv,piii,p6,p5,px` for 32 bit code

Flags for debugging aids

- -g generates symbolic debug information used by a debugger
- -gopt generates debug information in the presence of optimization
- -Mbounds adds array bounds checking
- -v gives verbose output, useful for debugging system or build problems
- -Minfo provides feedback on optimizations made by the compiler
- -S or –Mkeepasm to see the exact assembly generated

Go to Menu

Useful Compiler Flags

General

Flag	Comments
`-mp=nonuma`	Compile multithreaded code using OpenMP directives

Debugging

Flag	Comments
`-g`	For debugging symbols; put first
`-Ktrap=fp`	Trap floating point exceptions
`-Mchkptr`	Checks for unintended dereferencing of null pointers

Go to Menu

Optimization-Related PGI Compiler Options

- The compilers optimize code according to the specified optimization level. You can use a number of options to specify the optimization levels, including –O, –Mvect, –Mipa, and –Mconcur. In addition, you can use several of the –M<pgflag> switches to control specific types of optimization and parallelization.
- The optimization options are:

–fast	–Minline	–Mpfi	–Mvect
–Mconcur	–Mipa=fast	–Mpfo	–O
–Minfo	–Mneginfo	–Munroll	–Msafeptr

Go to Menu

Optimization-Related PGI Compiler Options (continued)

Option	Description
`-fast`	Generally optimal set of flags for targets that support SSE capability.
`-fastsse`	Generally optimal set of flags for targets that include SSE/SSE2 capability.
`-M<pgflag>`	Selects variations for code generation and optimization.
`-mp[=all, align, bind, [no]numa]`	Interpret and process user-inserted shared-memory parallel programming directives.
`-O<level>`	Specifies code optimization level where <level> is 0, 1, 2, 3, or 4.
`-pc <val>`	(–tp px/p5/p6/piii targets only) Set precision globally for x87 floating-point calculations; must be used when compiling the main program. <val> may be one of 32, 64 or 80.
`-Mprof=time`	Instrument the generated executable to produce a gprof-style

Optimization-Related PGI Compiler Options (continued)

- Traditional optimization controlled through -O[<n>], n is 0 to 4.
- -fastsse and -fast are equal to -O2 -Munroll=c:1 -Mnoframe –Mlre -Mvect=sse, -Mscalarsse -Mcache_align -Mflushz
  - For -Munroll, c specifies completely unroll loops with this loop count or less
  - -Munroll=n:<m> says unroll other loops m times
- -Mcache_align aligns top level arrays and objects on cache-line boundaries
- -Mflushz flushes SSE denormal numbers to zero
- -Mnoframe does not set up a stack frame
- -Mlre is loop-carried redundancy elimination

Outline: Getting Started with PGI Compiler Optimizations

- Quick Start
- Options for Getting Help
- Options for Getting Information
- Common Performance Challenges
- What is Vectorization on x64 CPUs?
- Optimization Strategies

Go to Menu

Quick Start

- To get started quickly with optimization, a good set of options to use with any of the PGI compilers is –fast –Mipa=fast. For example:

  ```
  $ ftn -fast -Mipa=fast prog.f
  ```

- For all of the PGI Fortran, C, and C++ compilers, the –fast –Mipa=fast options generally produce code that is well-optimized without the possibility of significant slowdowns due to pathological cases.
  - The –fast option is an aggregate option that includes a number of individual PGI compiler options; which PGI compiler options are included depends on the target for which compilation is performed.
  - The –Mipa=fast option invokes interprocedural analysis including several IPA suboptions.
  - For C++ programs, add -Minline=levels:10 --no_exceptions as shown here:

    ```
    $ CC -fast -Mipa=fast -Minline=levels:10 --no_exceptions prog.cc
    ```

-help and -Minfo

–help

- You can see a specification of any command line option by invoking any of the PGI compilers with -help in combination with the option in question, without specifying any input files. For example, you might want information on -O:
- $ pgf95 -help –O
- Or you can see the full functionality of -help itself, which can return information on either an individual option or groups of options:
- $ pgf95 -help –help

–Minfo

- Used to display compile-time optimization listings.
- When this option is used, the PGI compilers issue informational messages to stderr as compilation proceeds. From these messages, you can determine which loops are
  - optimized using unrolling,
  - SSE instructions,
  - vectorization,
  - parallelization,
  - interprocedural optimizations
  - various miscellaneous optimizations.
  - you can also see where and whether functions are inlined.

Go to Menu

–Mneginfo and –dryrun

–Mneginfo

- Used to display informational messages listing why certain optimizations are inhibited.

–dryrun

- Can be useful as a diagnostic tool if you need to see the steps used by the compiler driver to preprocess, compile, assemble and link in the presence of a given set of command line inputs.
- If –dryrun option is specified, these steps will be printed to stderr but are not actually performed.
- For example, you can use this option to inspect the default and user-specified libraries that are searched during the link phase, and the order in which they are searched by the linker.

Go to Menu

Common Performance Challenges

- **Vectorization**
  - What is vectorization? Is my code vectorizing?
  - Conflicts with C++ and F90 “ease of use” programming techniques. C and C++ pointer issues that prevent vectorization.
- **Multi-core issues**
  - Memory bandwidth
  - MPI, OpenMP, and auto parallelization
- **IPA** – Interproceedural Analysis and Inlining
  - IPA and inline enabled libraries

Go to Menu

What is Vectorization on x64 CPUs?

- **By a Programmer**: writing or modifying algorithms and loops to enable or maximize generation of x64 packed Streaming SIMD Extensions (SSE) instructions by a vectorizing compiler
- **By a Compiler**: identifying and transforming loops to use packed SSE arithmetic instructions which operate on more than one data element per instruction
- For more information, please, refer to the "*Software Optimization Guide for AMD64 Processors*" at http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF

Optimization Strategies

- Establish a workload
- Optimization from the top-down
- Use of proper tools, methods
- Processor level optimizations, parallel methods
- Different flags/features for different types of code

Outline: Optimization Categories (Node Level Tuning)

- Local and Global Optimization
- Vectorization
- Interprocedural Analysis (IPA)
- Function Inlining
- SMP Parallelization
- Miscellaneous Optimizations

Go to Menu

Local and Global Optimization

Go to Menu

Local Optimization

- This optimization is performed on a block-by-block basis within a program's basic blocks. A basic block is a sequence of statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching, except at the end.
- The PGI compilers perform many types of local optimization including:
  - algebraic identity removal,
  - constant folding,
  - common sub-expression elimination,
  - redundant load and store elimination,
  - scheduling,
  - strength reduction,
  - peephole optimizations.

Global Optimization

- This optimization is performed on a program unit over all its basic blocks. The optimizer performs control-flow and data-flow analysis for an entire program unit. All loops, including those formed by IFs and GOTOs, are detected and optimized.
- Global optimization includes:
  - constant propagation,
  - copy propagation,
  - dead store elimination,
  - global register allocation,
  - invariant code motion,
  - induction variable elimination.

Local and Global Optimization using –O

Using the PGI compiler commands with the –Olevel option (the capital O is for Optimize), you can specify any of the following optimization levels:

–O0 Level zero specifies no optimization. A basic block is generated for each language statement.

–O1 Level one specifies local optimization. Scheduling of basic blocks is performed. Register allocation is performed.

–O2 Level two specifies global optimization. This level performs all level-one local optimization as well as level two global optimization. If optimization is specified on the command line without a level, level 2 is the default.

–O3 Level three specifies aggressive global optimization. This level performs all level-one and level-two optimizations and enables more aggressive hoisting and scalar replacement optimizations that may or may not be profitable.

–O4 Level four performs all level-one, level-two, and level-three optimizations and enables hoisting of guarded invariant floating point expressions.

Note: If you use the –O option to specify optimization and do not specify a level, then level-two optimization (–O2) is the default.

Go to Menu

Local and Global Optimization using –O (continued)

- You can explicitly select the optimization level on the command line. For example, the following command line specifies level-two optimization which results in global optimization:
- $ pgf95 -O2 prog.f
- Specifying –O on the command-line without a level designation is equivalent to –O2. The default optimization level changes depending on which options you select on the command line. For example, when you select the –g debugging option, the default optimization level is set to level-zero (–O0). However, if you need to debug optimized code, you can use the -gopt option to generate debug information without perturbing optimization.
- As noted previously, the –fast option includes –O2 on all x86 and x64 targets. If you want to override the default for–fast with –O3 while maintaining all other elements of –fast, simply compile as follows:
- $ pgf95 -fast -O3 prog.f

Go to Menu

Vectorization

Go to Menu

Loop Optimization: Unrolling, Vectorization, and Parallelization

- The performance of certain classes of loops may be improved through vectorization or unrolling options.
  - Vectorization transforms loops to improve memory access performance and make use of packed SSE instructions which perform the same operation on multiple data items concurrently.
  - Unrolling replicates the body of loops to reduce loop branching overhead and provide better opportunities for local optimization, vectorization and scheduling of instructions.
- Performance for loops on systems with multiple processors may also improve using the parallelization features of the PGI compilers.

Vectorizable F90 Array Syntax Data is REAL*4

```
350 !
351 !   Initialize vertex, similarity and coordinate arrays
352 !
353     Do Index = 1, NodeCount
354       IX = MOD (Index - 1, NodesX) + 1
355       IY = ((Index - 1) / NodesX) + 1
356       CoordX (IX, IY) = Position (1) + (IX - 1) * StepX
357       CoordY (IX, IY) = Position (2) + (IY - 1) * StepY
358       JetSim (Index)  = SUM (Graph (:, :, Index) * &
359     &                   GaborTrafo (:, :, CoordX(IX,IY), CoordY(IX,IY)))
360       VertexX (Index) = MOD (Params%Graph%RandomIndex (Index) - 1, NodesX) + 1
361       VertexY (Index) = ((Params%Graph%RandomIndex (Index) - 1) / NodesX) + 1
362     End Do
```

Inner “loop” at line 358 is vectorizable, can used packed SSE instructions

-fastsse -Minfo

–fastsse to Enable SSE Vectorization
–Minfo to List Optimizations to stderr

```
% pgf95 -fastsse -Mipa=fast -Minfo -S graphRoutines.f90
…
localmove:
   334, Loop unrolled 1 times (completely unrolled)
   343, Loop unrolled 2 times (completely unrolled)
   358, Generated an alternate loop for the inner loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
…
```

Scalar SSE vs. Vector SSE

Scalar SSE:

```
.LB6_668:
# lineno: 358
        movss   -12(%rax),%xmm2
        movss   -4(%rax),%xmm3
        subl    $1,%edx
        mulss   -12(%rcx),%xmm2
        addss   %xmm0,%xmm2
        mulss   -4(%rcx),%xmm3
        movss   -8(%rax),%xmm0
        mulss   -8(%rcx),%xmm0
        addss   %xmm0,%xmm2
        movss   (%rax),%xmm0
        addq    $16,%rax
        addss   %xmm3,%xmm2
        mulss   (%rcx),%xmm0
        addq    $16,%rcx
        testl   %edx,%edx
        addss   %xmm0,%xmm2
        movaps  %xmm2,%xmm0
        jg      .LB6_625
```

Vector SSE:

```
.LB6_1245:
# lineno: 358
        movlps  (%rdx,%rcx),%xmm2
        subl    $8,%eax
        movlps  16(%rcx,%rdx),%xmm3
        prefetcht0  64(%rcx,%rsi)
        prefetcht0  64(%rcx,%rdx)
        movhps  8(%rcx,%rdx),%xmm2
        mulps   (%rsi,%rcx),%xmm2
        movhps  24(%rcx,%rdx),%xmm3
        addps   %xmm2,%xmm0
        mulps   16(%rcx,%rsi),%xmm3
        addq    $32,%rcx
        testl   %eax,%eax
        addps   %xmm3,%xmm0
        jg      .LB6_1245:
```

Facerec Scalar: 104.2 sec

Facerec Vector: 84.3 sec

Go to Menu

Vectorizable C Code Fragment?

```
217   void func4(float *u1, float *u2, float *u3, …
      …
221   for (i = -NE+1, p1 = u2-ny, p2 = n2+ny; i < nx+NE-1; i++)
222         u3[i] += clz * (p1[i] + p2[i]);
223   for (i = -NI+1, i < nx+NE-1; i++) {
224         float vdt = v[i] * dt;
225         u3[i] = 2.*u2[i]-u1[i]+vdt*vdt*u3[i];
226   }
```

```
% pgcc –fastsse –Minfo functions.c
func4:
    221, Loop unrolled 4 times
    221, Loop not vectorized due to data dependency
    223, Loop not vectorized due to data dependency
```

Go to Menu

Pointer Arguments Inhibit Vectorization

```
217   void func4(float *u1, float *u2, float *u3, …
      …
221   for (i = -NE+1, p1 = u2-ny, p2 = n2+ny; i < nx+NE-1; i++)
222         u3[i] += clz * (p1[i] + p2[i]);
223   for (i = -NI+1, i < nx+NE-1; i++) {
224         float vdt = v[i] * dt;
225         u3[i] = 2.*u2[i]-u1[i]+vdt*vdt*u3[i];
226   }
```

```
% pgcc –fastsse –Msafeptr –Minfo functions.c
func4:
    221, Generated vector SSE code for inner loop
         Generated 3 prefetch instructions for this loop
    223, Unrolled inner loop 4 times
```

Go to Menu

C Constant Inhibits Vectorization

```
217   void func4(float *u1, float *u2, float *u3, …
      …
221   for (i = -NE+1, p1 = u2-ny, p2 = n2+ny; i < nx+NE-1; i++)
222         u3[i] += clz * (p1[i] + p2[i]);
223   for (i = -NI+1, i < nx+NE-1; i++) {
224         float vdt = v[i] * dt;
225         u3[i] = 2.*u2[i]-u1[i]+vdt*vdt*u3[i];
226   }
```

```
% pgcc –fastsse –Msafeptr –Mfcon –Minfo functions.c
func4:
    221, Generated vector SSE code for inner loop
         Generated 3 prefetch instructions for this loop
    223, Generated vector SSE code for inner loop
         Generated 4 prefetch instructions for this loop
```

-Msafeptr Option and Pragma

–M[no]safeptr[=all | arg | auto | dummy | local | static | global]

all	all pointers are safe
arg	argument pointers are safe
local	local pointers are safe
static	static local pointers are safe
global	global pointers are safe

#pragma [*scope*] [no]safeptr={arg | local | global | static | all},…

Where *scope* is *global*, *routine* or *loop*

Common Barriers to SSE Vectorization

Potential Dependencies & C Pointers – Give compiler more info with –Msafeptr, pragmas, or restrict type qualifer

Function Calls – Try inlining with –Minline or –Mipa=inline

Type conversions – manually convert constants or use flags

Too few iterations – Usually better to unroll the loop

Real dependencies – Must restructure loop, if possible

Go to Menu

Barriers to Efficient Execution of Vector SSE Loops

- Not enough work – vectors are too short
- Vectors not aligned to a cache line boundary
- Non-unity strides
- May run out of space to handle all the instructions
- **Code bloat if altcode is generated**

Go to Menu

Interprocedural Analysis (IPA)

Go to Menu

Interprocedural Analysis (IPA) and Optimization

- Interprocedural analysis (IPA) allows use of information across function call boundaries to perform optimizations that would otherwise be unavailable. For example, if the actual argument to a function is in fact a constant in the caller, it may be possible to propagate that constant into the callee and perform optimizations that are not valid if the dummy argument is treated as a variable.
- A wide range of optimizations are enabled or improved by using IPA, including but not limited to
  - data alignment optimizations,
  - argument removal,
  - constant propagation,
  - pointer disambiguation,
  - pure function detection,
  - F90/F95 array shape propagation,
  - data placement,
  - vestigial function removal,
  - automatic function inlining,
  - inlining of functions from pre-compiled libraries,
  - interprocedural optimization of functions from pre-compiled libraries.

Go to Menu

What can Interprocedural Analysis and Optimization with –Mipa do for You?

- Interprocedural constant propagation
- Pointer disambiguation
- Alignment detection, Alignment propagation
- Global variable mod/ref detection
- F90 shape propagation
- Function inlining

Go to Menu

Effect of IPA on the WUPWISE Benchmark

PGF95 Compiler Options	Execution Time in Seconds
–fastsse	156.49
–fastsse –Mipa=fast	121.65
–fastsse –Mipa=fast,inline	91.72

- –Mipa=fast => constant propagation => compiler sees complex matrices are all 4x3 => completely unrolls loops

- –Mipa=fast,inline => small matrix multiplies are all inlined

Go to Menu

Using Interprocedural Analysis

- Must be used at both compile time and link time
- Non-disruptive to development process – edit/build/run
- Speed-ups of 5% - 10% are common
- –Mipa=safe:<*name*> - safe to optimize functions which call or are called from unknown function/library *name*
- –Mipa=libopt – perform IPA optimizations on libraries
- –Mipa=libinline – perform IPA inlining from libraries

Go to Menu

Function Inlining

Go to Menu

Function Inlining

- This optimization allows a call to a function to be replaced by a copy of the body of that function. This optimization will sometimes speed up execution by eliminating the function call and return overhead.
- Function inlining may also create opportunities for other types of optimization.
- Function inlining is not always beneficial.
- When used improperly it may increase code size and generate less efficient code.

Go to Menu

Explicit Function Inlining

–Minline[=[lib:]<inlib> | [name:]<func> | except:<func> | size:<n> | levels:<n>]

[lib:]<inlib>	Inline extracted functions from *inlib*
[name:]<func>	Inline function func
except:<func>	Do not inline function func
size:<n>	Inline only functions smaller than n statements (approximate)
levels:<n>	Inline n levels of functions

For C++ Codes, PGI Recommends IPA-based inlining or –Minline=levels:10!

Specific recommendations for C++

- Encapsulation; Data hiding
  - small functions, inline!
- Exception handling
  - use `-no_exceptions` until 7.0
- Overloaded operators, overloaded functions
  - Can be used
- Pointer Chasing
  - `-Msafeptr`, restrict qualifier
- Templates, Generic Programming
  - Can be used. However, aggressive use of templates may still run into problems
- Inheritance, polymorphism, virtual functions
  - runtime lookup or check, no inlining, potential performance penalties

SMP Parallelization

Go to Menu

SMP - Shared-Memory Parallel

- The PGI compilers support two styles of SMP parallel programming:
  - Automatic shared-memory parallel programs compiled using the `-Mconcur` option to pgf77, pgf95, pgcc, or pgcpp - parallel programs of this variety can be run on shared-memory parallel (SMP) systems such as dual-core or multi-processor workstations.
  - OpenMP shared-memory parallel programs compiled using the `-mp` option to pgf77, pgf95, pgcc, or pgcpp - parallel programs of this variety can be run on SMP systems. Carefully coded user-directed parallel programs using OpenMP directives can often achieve significant speed-ups on dual-core workstations or large numbers of processors on SMP server systems.
- OpenMP parallel programs can only work within one node.

SMP Parallelization

- Use **-mp** to enable OpenMP parallel programming model
- OpenMP programs compiled without –mp option will still work ignoring OpenMP-related lines in the code.
- Caution: Yes, they work, but they may not give the best performance. For example, a programmer still may need to analyze and rewrite the loop.

- See PGI User's Guide at
  - http://www.pgroup.com/doc/pgiug.pdf , or
- OpenMP Specifications/Documentation at
  - http://openmp.org/wp/openmp-specifications/
- OpenMP tutorial from LLNL
  - https://computing.llnl.gov/tutorials/openMP/

Calculation of Overhead

- The set of microbenchmarks can be used to measure the overheads of synchronization, loop scheduling and array operations in the OpenMP runtime library:
  - http://www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_index.html
- Synchronization benchmark measures the overhead incurred by the following directives, all of which contain barrier synchronization: PARALLEL(with and without a REDUCTION clause), DO/*for*, PARALLEL DO/*parallel for*, BARRIER, SINGLE.
- To measure the overhead of the PARALLEL directive, for example, the time taken for overhead computed as (mean) difference in execution time:

```
do i=1,reps
        call dummy()
end do
```

is subtracted from the time taken for

```
do i=1,reps
!$OMP PARALLEL
        call dummy()
!$OMP END PARALLEL
end do
```

Calculation of Overhead (continued)

- The array benchmark compares the overheads associated with various Data Scope Attribute Clauses when applied to arrays. The clauses considered are: PRIVATE, FIRSTPRIVATE, COPYPRIVATE, COPYIN and REDUCTION. The variation of overheads with array size is examined.
- To measure the overhead of the COPYIN array directive, for example, the time taken for

```
do i=1,reps
      call dummy(a)
end do
```

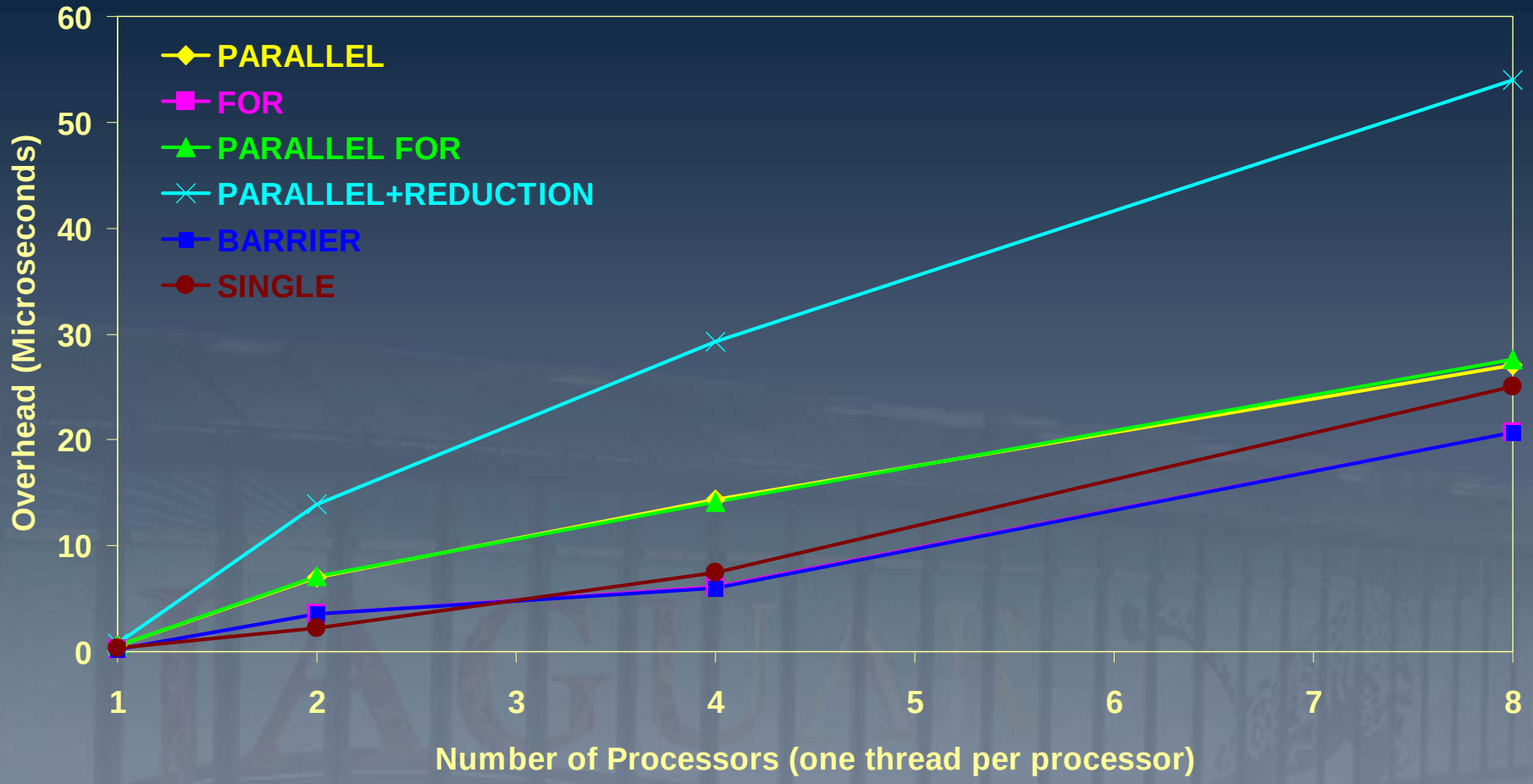
is subtracted from the time taken for

```
do i=1,reps
!$OMP PARALLEL COPYIN(a)
      call dummy(a)
!$OMP END PARALLEL
end do
```

where `dummy` is a routine which contains a dummy loop performing simple operations on the array `a`. For the COPYIN and COPYPRIVATE clauses, the array passed to dummy must be declared as THREADPRIVATE.

Go to Menu

Synchronization benchmark (Cray XT5)

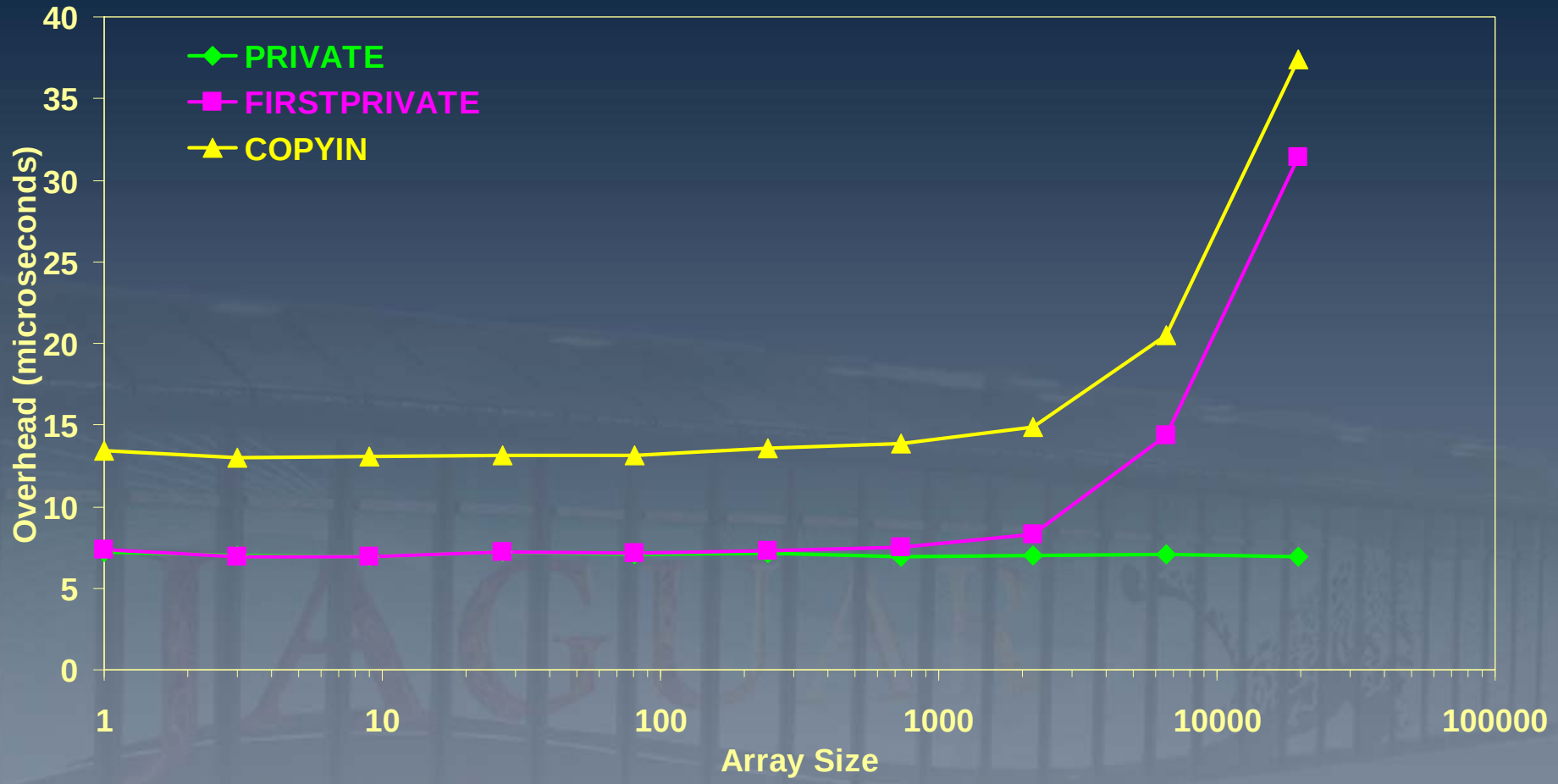


Note:

- PARALLEL directive line is located behind PARALLEL FOR
- FOR directive line is located behind BARRIER

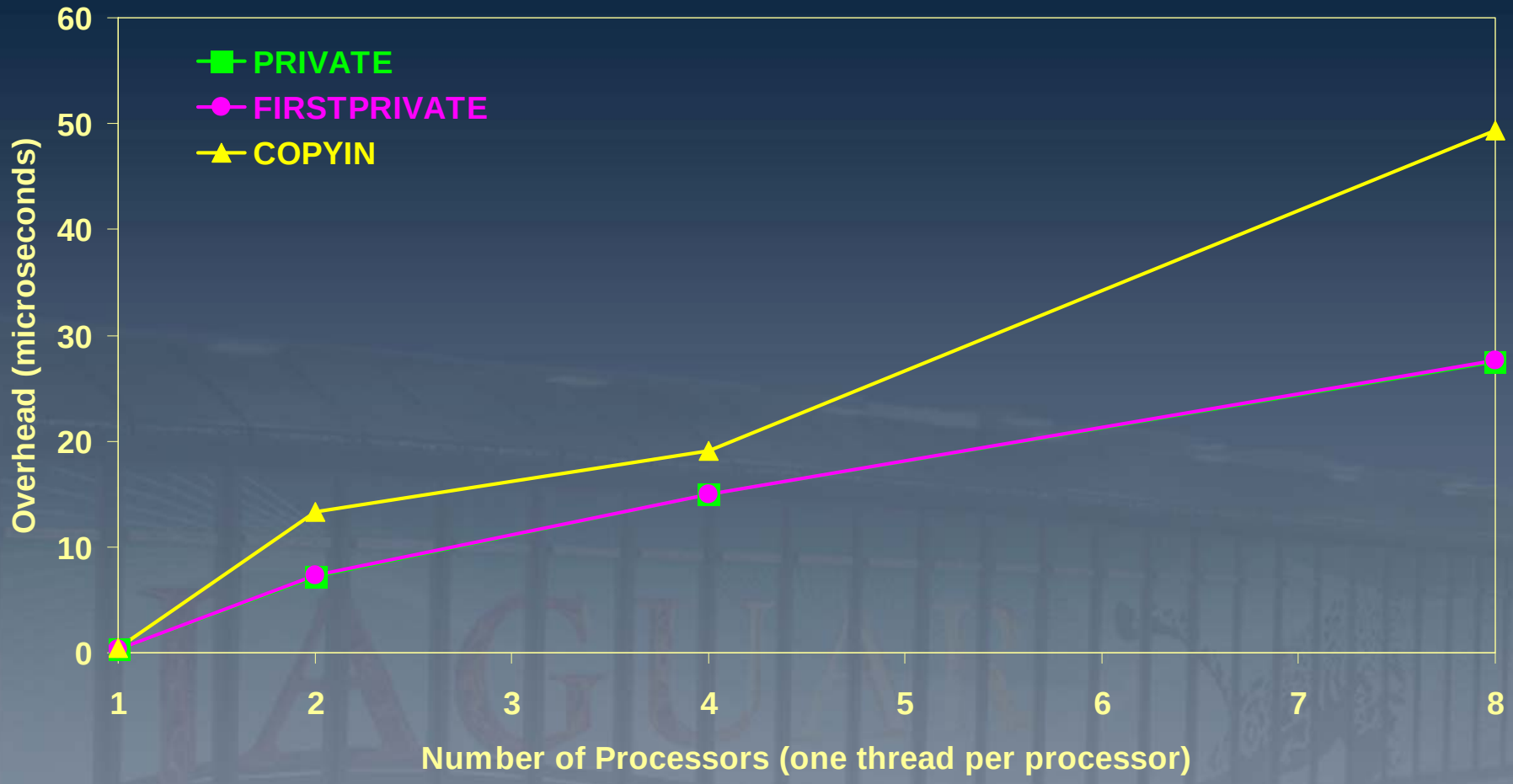
Go to Menu

Array benchmark – Two Threads (Cray XT5)



Go to Menu

Array benchmark – Array of Size a[1] (Cray XT5)



Note:

- PRIVATE clause line is located behind FIRSTPRIVATE

Go to Menu

Miscellaneous Optimizations

Go to Menu

Profile-Feedback Optimization (PFO)

- Profile-feedback optimization (PFO) makes use of information from a trace file produced by specially instrumented executables which capture and save information on
  - branch frequency,
  - function and subroutine call frequency,
  - semi-invariant values,
  - loop index ranges,
  - other input data dependent information that can only be collected dynamically during execution of a program.
- By definition, use of profile-feedback optimization is a two-phase process: compilation and execution of a specially-instrumented executable, followed by a subsequent compilation which reads a trace file generated during the first phase and uses the information in that trace file to guide compiler optimizations.

Go to Menu

Miscellaneous Optimizations

- –Mfprelaxed
  - single-precision sqrt, rsqrt, div performed using reduced-precision reciprocal approximation.
  - *Caution: This should only be used if the code can tolerate a loss of precision (2-3 decimal points)*
- –Mprefetch=d:<p>,n:<q>
  - control prefetching distance, max number of prefetch instructions per loop
- –M[no]movnt
  - disable / force non-temporal moves
- –V[version]
  - to switch between PGI releases at file level
- –Mvect=noaltcode
  - disable multiple versions of loops

PGI Documentation and Support

- The Portland Group website
  - http://www.pgroup.com/
- PGI provided documentation
  - http://www.pgroup.com/resources/docs.htm
- PGI User Forums
  - https://www.pgroup.com/userforum/index.php
- PGI FAQs, Tips & Techniques pages

Cray X86 compilers

- Cray provides its own Cray X86 high-performance compiler set as part of several programming environments on Jaguar.
- To switch to Cray X86 compilers from PGI compilers loaded by default, please, refer to the Introduction section of this document.

Outline: Getting Started with Cray Compiler Optimizations

- Quick Start
- Directives
- Current Strengths

Quick Start

- Make sure it is available
  - module avail PrgEnv-cray
- To access the Cray compiler
  - module load PrgEnv-cray
- To target the various chips
  - module load xtpe-barcelona,shanghi,istanbul]
- Once you have loaded the module “cc” and “ftn” are the Cray compilers
  - Recommend just using default options
  - Use –rm (fortran) and –hlist=m (C) to find out what happened
- Example: ftn –rm –c file.f90

Directives

- Cray compiler supports a full and growing set of directives and pragmas
  - !dir$ concurrent
  - !dir$ ivdep
  - !dir$ interchange
  - !dir$ unroll
  - !dir$ loop_info [max_trips] [cache_na] ... Many more
  - !dir$ blockable
- man directives
- man loop_info

Current Strengths

- Loop Based Optimizations
  - Vectorization
  - Interchange
  - Pattern Matching
  - Cache blocking/ non-temporal / prefetching
- Fortran Standard
- PGAS (UPC and Co-Array Fortran)
- Optimization Feedback: Loopmark
- Focus

Go to Menu

Outline: Optimization Options

- General Optimization Options
- Automatic Cache Management Options
- Vector Optimization Options
- Inlining Optimization Options
- Scalar Optimization Options
- Math Options
- Floating-point Optimization Levels

Go to Menu

General Optimization Options

`-O level`

Default: Equivalent to the appropriate -h option except that -O3 is equivalent to -h cache2

The -O *level* option specifies the optimization level for a group of compiler features. Specifying -O with no argument is the same as not specifying the -O option; this syntax is supported for compatibility with other vendors.

A value of 0, 1, 2, or 3 sets that level of optimization for each of the -h scalar*n* and -h vector*n* options.

The -O values of 0, 1, 2, or 3 set that level of optimization for -h cache*n* options, except that -O3 is equivalent to -h cache2.

The -O2 option is equivalent to ipa2, scalar2, vector2, cache2, and thread2.

Optimization features specified by -O are equivalent to the -h options(`-h cache`*n;* `-h vector`*n* `-h scalar`*n)* discussed below

General Optimization Options (continued)

`-h [no]aggress`

Default: -h noaggress

The -h aggress option provides greater opportunity to optimize loops that would otherwise by inhibited from optimization due to an internal compiler size limitation. -h noaggress leaves this size limitation in effect.

With -h aggress, internal compiler tables are expanded to accommodate larger loop bodies. This option can increase the compilation's time and memory size.

`-h [no]autothread`

Default: -h noautothread

The -h [no]autothread option enables or disables automatic threading.

`-h display_opt`

The -h display_opt option displays the current optimization settings for this compilation.

`-h [no]dwarf`

Default: -h dwarf

The -h [no]dwarf option controls whether DWARF debugging information is generated during compilation.

Go to Menu

General Optimization Options (continued)

`-h fusion`*n*

Default: -h fusion2

The –h fusion*n* option controls loop fusion and changes the assertiveness of the fusion pragma. Loop fusion can improve the performance of loops, although in rare cases it may degrade performance. The *n* argument allows you to turn loop fusion on or off and determine where fusion should occur.

Note: Loop fusion is disabled when the scalar level is set to 0.

The values for *n* are:

0 No fusion (ignore all fusion pragmas and do not attempt to fuse other loops)

1 Attempt to fuse loops that are marked by the fusion pragma.

2 Attempt to fuse all loops (includes array syntax implied loops), except those marked with the nofusion pragma.

`-h [no]intrinsics`

Default: -h intrinsics

The -h intrinsics option allows the use of intrinsic hardware functions, which allow direct access to some hardware instructions or generate inline code for some functions. This option has no effect on specially handled library functions.

General Optimization Options (continued)

`-h list`

The -h list=*opt* option allows you to create listings and control their formats. The listings are written to *source_file_name_without_suffix*.lst. The values for *opt* are:

a Use all list options; *source_file_name_without_suffix*.lst includes a summary report, an options report, and the source listing.

d Decompiles (translates) the intermediate representation of the compiler into listings that resemble the format of the source code. This is performed twice, resulting in two output files, at different points during the optimization process. You can use these files to examine the restructuring and optimization changes made by the compiler, which can lead to insights about changes you can make to your source code to improve its performance.

e Expand include files.

Note: Using this option may result in a very large listing file. All system include files are also expanded.

i Intersperse optimization messages within the source listing rather than at the end.

m Create loopmark listing; *source_file_name_without_suffix*.lst includes summary report and source listing.

s Create a complete source listing (include files not expanded). Using -h list=m creates a loopmark listing. The e, i, s, and w options provide additional listing features. Using -h list=a combines all options.

General Optimization Options (continued)

`-h [no]msgs`

Default: -h nomsgs

The -h msgs option causes the compiler to write optimization messages to stderr. When the -h msgs option is in effect, you may request that a listing be produced so that you can see the optimization messages in the listing.

`-h [no]negmsgs`

Default: -h nonegmsgs

The -h negmsgs option causes the compiler to generate messages to stderr that indicate why optimizations such as vectorization or inlining did not occur in a given instance. The -h negmsgs option enables the -h msgs option. The -h list=a option enables the -h negmsgs option.

`-h [no]omp_trace`

Default: -h noomp_trace (tracing is off)

The -h [no]omp_trace option turns the insertion of the CrayPat OpenMP tracing calls on or off.

`-h [no]func_trace`

The -h func_trace option is for use only with CrayPat. If this option is specified, the compiler inserts CrayPat trace entry points into each function in the compiled source file. The names of the trace entry points are:

- __pat_tp_func_entry
- __pat_tp_func_return

These are resolved by CrayPat when the program is instrumented using the pat_build command. When the instrumented program is executed and it encounters either of these trace entry points, CrayPat captures the address of the current function and its return address.

Go to Menu

General Optimization Options (continued)

`-h [no]overindex`

Default: -h nooverindex

The -h overindex option declares that there are array subscripts that index a dimension of an array that is outside the declared bounds of that array. The -h nooverindex option declares that there are no array subscripts that index a dimension of an array that is outside the declared bounds of that array.

`-h [no]pattern`

Default: -h pattern

The -h [no]pattern option globally enables or disables pattern matching. When the compiler recognizes certain patterns in the source code, it replaces the construct with a call to an optimized library routine. A loop or statement that has been pattern matched and replaced with a call to a library routine is indicated with an A in the loopmark listing.

Note: Pattern matching is not always worthwhile. If there is a small amount of work in the pattern-matched construct, the call overhead may outweigh the time saved by using the optimized library routine. When compiling using the default optimization settings, the compiler attempts to determine whether each given candidate for pattern matching will in fact yield improved performance.

`-h profile_generate`

The -h profile_generate option directs that the source code be instrumented for gathering profile information. The compiler inserts calls and data-gathering instructions to allow CrayPat to gather information about the loops in a compilation unit. If you use this option, you must run CrayPat on the resulting executable so the CrayPat data-gathering routines are linked in.

General Optimization Options (continued)

`-h thread`*n*

Default: –h thread2

The -h thread*n* options control the optimization of both OpenMP and automatic threading. The values of *n* are:

0 No autothreading or OMP (OpenMP) threading.

1 No parallel region expansion, no loop restructuring for OMP loops, no optimization across OMP constructs.

2 Parallel region expansion, limited loop restructuring, optimization across OMP constructs.

3 Reduction results may not be repeatable. Loop restructuring, including modifying iteration space for static schedules (breaking standard compliance).

`-h unroll`*n*

Default: –h unroll2

The -h unroll*n* option globally controls loop unrolling and changes the assertiveness of the unroll pragma. By default, the compiler attempts to unroll all loops, unless the nounroll pragma is specified for a loop. Generally, unrolling loops increases single processor performance at the cost of increased compile time and code size. The *n* argument allows you to turn loop unrolling on or off and specify where unrolling should occur. It also affects the assertiveness of the unroll pragma. The values for *n* are:

0 No unrolling (ignore all unroll pragmas and do not attempt to unroll other loops).

1 Attempt to unroll loops that are marked by the unroll pragma.

2 Unroll loops when performance is expected to improve. Loops marked with the unroll or nounroll pragma override automatic unrolling.

Note: Loop unrolling is disabled when the scalar level is set to 0.

Go to Menu

Automatic Cache Management Options

`-h cachen`

Default: -h cache2

The -h cache*n* option specifies the levels of automatic cache management to perform. The default is -h cache2. The values for *n* are:

0 Cache blocking (including directive-based blocking) is turned off. This level is compatible with all scalar and vector optimization levels.

1 Conservative automatic cache management. Characteristics include moderate compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the predicted cache footprint of the symbol in isolation is small enough to experience the reuse.

2 Moderately aggressive automatic cache management. Characteristics include moderate compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the predicted state of the cache model is such that the symbol will experience the reuse.

3 Aggressive automatic cache management. Characteristics include potentially high compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the allocation of the symbol to the cache is predicted to increase the number of cache hits.

-O Option	**Cache Level**
`-O0`	`-h cache0`
`-O1`	`-h cache1`
`-O2`	`-h cache2`

Go to Menu

Vector Optimization Options

`-h vector`*n*

Default: -h vector2

The -h vector*n* option specifies the level of automatic vectorizing to be performed. Vectorization results in significant performance improvements with a small increase in object code size. Vectorization directives are unaffected by this option. The values of *n* are:

n	Description
0	No automatic vectorization. Characteristics include low compile time and small compile size. This option is compatible with all scalar optimization levels.
1	Specifies conservative vectorization. Characteristics include moderate compile time and size. No loop nests are restructured; only inner loops are vectorized. No vectorizations that might create false exceptions are performed. Results may differ slightly from results obtained when -h vector0 is specified because of vector reductions. The -h vector1 option is compatible with -h scalar1, -h scalar2, and -h scalar3.
2	Specifies moderate vectorization. Characteristics include moderate compile time and size. Loop nests are restructured. The -h vector2 option is compatible with -h scalar2 and -h scalar3.
3	Specifies aggressive vectorization. Characteristics include potentially high compile time and size. Loop nests are restructured. Vectorizations that might create false exceptions in rare cases may be performed.

Go to Menu

Inlining Optimization Options

`-h ipa`*n*

Default: -h ipa3

The -h ipa*n* option allows the compiler to automatically decide which procedures to consider for inlining. Procedures that are potential targets for inline expansion include all the procedures within the input file to the compilation. Table below explains what is inlined at each level.

Inlining level	Description
0	All inlining is disabled. All inlining compiler directives are ignored.
1	Directive inlining. Inlining is attempted for call sites and routines that are under the control of an inlining pragma directive.
2	Call nest inlining. Inline a call nest to an arbitrary depth as long as the nest does not exceed some compiler-determined threshold. A call nest can be a leaf routine. The expansion of the call nest must yield straight-line code (code containing no external calls) for any expansion to occur.
3	Constant actual argument inlining. This includes levels 1 and 2, plus any call site that contains a constant actual argument. This is the default inlining level.
4	Tiny routine inlining plus cloning. This includes levels 1, 2, and 3, plus the inlining of very small routines, regardless of where those routines fall in the call graph. The lower limit threshold is an internal compiler parameter. Also, routine cloning is attempted if inlining fails at a given call site.

Inlining Optimization Options (continued)

`-h ipafrom=`*`source`* `[`*`source`*`] ...`

The -h ipafrom=*source* [:*source*] option allows you to explicitly indicate the procedures to consider for inline expansion. The *source* arguments identify each file or directory that contains the routines to consider for inlining. Whenever a call is encountered in the input program that matches a routine in *source*, inlining is attempted for that call site.

Note: Spaces are not allowed on either side of the equal sign.

All inlining directives are recognized with explicit inlining. For information about inlining directives, see Inlining Directives on page 89.

Note: The routines in *source* are not actually loaded with the final program. They are simply templates for the inliner. To have a routine contained in *source* loaded with the program, you must include it in an input file to the compilation.

Scalar Optimization Options

`-h [no]interchange`

Default: -h interchange

- The -h interchange option allows the compiler to attempt to interchange all loops, a technique that is used to gain performance by having the compiler swap an inner loop with an outer loop. The compiler attempts the interchange only if the interchange will increase performance. Loop interchange is performed only at scalar optimization level 2 or higher.
- The -h nointerchange option prevents the compiler from attempting to interchange any loops. To disable interchange of loops individually, use the

  #pragma CRI nointerchange directive.

Scalar Optimization Options (continued)

`-h scalar`*n*

Default: -h scalar2

The -h scalar*n* option specifies the level of automatic scalar optimization to be performed. Scalar optimization directives are unaffected by this option. The values for *n* are:

0 Minimal automatic scalar optimization. The -h matherror=errno and the -h zeroinc options are implied by -h scalar0.

1 Conservative automatic scalar optimization. This level implies -h matherror=abort and -h nozeroinc.

2 Aggressive automatic scalar optimization. The scalar optimizations that provide the best application performance are used, with some limitations imposed to allow for faster compilation times.

3 Very aggressive optimization; compilation times may increase significantly.

`-h [no]zeroinc`

Default: -h nozeroinc

The -h nozeroinc option improves run time performance by causing the compiler to assume that constant increment variables (CIVs) in loops are not incremented by expressions with a value of 0. The -h zeroinc option causes the compiler to assume that some constant increment variables (CIVs) in loops might be incremented by 0 for each pass through the loop, preventing generation of optimized code. For example, in a loop with index *i*, the expression *expr* in the statement *i* +=*expr* can evaluate to 0. This rarely happens in actual code. -h zeroinc is the safer and slower option. This option is affected by the -h scalar*n* option

Go to Menu

Math Options

`-h fp`*n*

Default: -h fp2

The -h fp option allows you to control the level of floating-point optimizations. The *n* argument controls the level of allowable optimization; 0 gives the compiler minimum freedom to optimize floating-point operations, while 3 gives it maximum freedom. The higher the level, the lesser the floating-point operations conform to the IEEE standard. This option is useful for code using algorithms that are unstable but optimizable. Generally, this is the behavior and usage for each -h fp level:

- The -h fp0 option causes your program's executable code to conform more closely to the IEEE floating-point standard than the default mode (-h fp2). When you specify this level, many identity optimizations are disabled, vectorization of floating-point reductions are disabled, executable code is slower than higher floating-point optimization levels, and a scaled complex divide mechanism is enabled that increases the range of complex values that can be handled without producing an underflow.
- **Note:** Use the -h fp0 option only when your code pushes the limits of IEEE accuracy or requires strong IEEE standard conformance.
- The -h fp1 option performs various, generally safe, non-conforming IEEE optimizations, such as foldinga == ato true, where a is a floating point object. At this level, floating-point reassociation1 is greatly limited, which may affect the performance of your code. You should never use the -h fp1 option except when your code pushes the limits of IEEE accuracy or requires strong IEEE standard conformance.
- -h fp2 — includes optimizations of -h fp1.
- -h fp3 — includes optimizations of -h fp2.
- You should use the -h fp3 option when performance is more critical than the level of IEEE standard conformance provided by -h fp2.

Go to Menu

Floating-point Optimization Levels

Table below compares the various optimization levels of the -h fp option (levels 2 and 3 are usually the same). The table lists some of the optimizations performed; the compiler may perform other optimizations not listed.

Optimization Type	fp0	fp1	fp2 (default)	fp3
Complex divisions	Accurate and slower	Accurate and slower	Less accurate (less precision) and faster.	Less accurate (less precision) and faster.
Exponentiation rewrite	None	None	Maximum performance	Maximum performance
Strength reduction	Fast	Fast	Aggressive	Aggressive
Rewrite division as reciprocal equivalent	None	None	Yes	Aggressive
Floating point reductions	Slow	Fast	Fast	Fast
Safety	Maximum	Moderate	Moderate	Low
Expression factoring	None	Yes	Yes	Yes
Expression tree balancing	None	No	Yes	Yes

Loopmark: Compiler Feedback

- Compiler can generate an **filename.lst** file.
  - Contains annotated listing of your source code with letter indicating important optimizations

%%% Loopmark Legend %%%

Primary loop type	Modifiers
	a – vector atomic memory operation
A – Pattern matched	**b – blocked**
C – Collapsed	f – fused
D – Deleted	m – streamed but not partitioned
I – Inlined	p – conditional, partial and/or computed
M – Multithreaded	**r – unrolled**
P – Parallel/Tasked	s – shortloop
V – Vectorized	t – array syntax temp used
W – Unwound	w – unwound

Go to Menu

Example: Cray loopmark messages for Resid

- ftn –rm … or cc –hlist=m …

```
29. b-------< do i3=2,n3-1
30. b b-----< do i2=2,n2-1
31. b b Vr--< do i1=1,n1
32. b b Vr u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)
33. b b Vr > + u(i1,i2,i3-1) + u(i1,i2,i3+1)
34. b b Vr u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
35. b b Vr > + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
36. b b Vr--> enddo
37. b b Vr--< do i1=2,n1-1
38. b b Vr r(i1,i2,i3) = v(i1,i2,i3)
39. b b Vr > - a(0) * u(i1,i2,i3)
40. b b Vr > - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) )
41. b b Vr > - a(3) * ( u2(i1-1) + u2(i1+1) )
42. b b Vr--> enddo
43. b b-----> enddo
44. b-------> enddo
```

Example: Cray loopmark messages for Resid (continued)

- ftn-6289 ftn: VECTOR File = resid.f, Line = 29
  - A loop starting at **line 29 was not vectorized** because a recurrence was found on "U1" between lines 32 and 38.
- ftn-6049 ftn: SCALAR File = resid.f, Line = 29
  - A loop starting **at line 29 was blocked with block size 4.**
- ftn-6289 ftn: VECTOR File = resid.f, Line = 30
  - A loop starting at line 30 was not vectorized because a recurrence was found on "U1" between lines 32 and 38.
- ftn-6049 ftn: SCALAR File = resid.f, Line = 30
  - A loop starting at line 30 was blocked with block size 4.
- ftn-6005 ftn: SCALAR File = resid.f, Line = 31
  - A loop starting at **line 31 was unrolled 4 times.**
- ftn-6204 ftn: VECTOR File = resid.f, Line = 31
  - A loop starting at **line 31 was vectorized.**
- ftn-6005 ftn: SCALAR File = resid.f, Line = 37
  - A loop starting at line 37 was unrolled 4 times.
- ftn-6204 ftn: VECTOR File = resid.f, Line = 37
  - A loop starting at **line 37 was vectorized.**

Go to Menu

Byte Swapping

- `-hbyteswapio`
  - Link time option
  - Applies to all unformatted fortran IO
- Assign command
  - With the PrgEnv-cray module loaded do this:
    - `setenv FILENV assign.txt`
    - `setenv –N swap_endian g:su`
    - `setenv –N swap_endian g:du`
- Can use assign to be more precise

Cray X86 Related Publications

- The following documents contain additional information that may be helpful:
  - *Cray C and C++ Reference Manual (http://docs.cray.com)*
  - cc(1) compiler driver man page for all Cray XT C compilers
  - craycc(1) man page for the Cray C compiler
  - CC(1) compiler driver man page for all Cray XT C++ compilers
  - crayCC(1) man page for the Cray C++ compiler
  - intro_pragmas(1) man page
  - *Cray Fortran Reference Manual (http://docs.cray.com)*
  - ftn(1) compiler driver man page for all Cray XT Fortran compilers
  - crayftn(1) man page for the Cray Fortran compiler
  - *Cray XT Programming Environment User's Guide (http://docs.cray.com)*
  - aprun(1) man page
  - *Using Cray Performance Analysis Tools*
  - *Cray Application Developer's Environment Installation Guide (http://docs.cray.com)*

Outline: Resources for Users

- Optimization-Related References
- Getting Started
- Advanced Topics
- More Information

Resources for Users: Optimization-Related References

- *Software Optimization Guide for AMD64 Processors* (Guidelines for serial optimizations specific to AMD Opteron on the AMD site):
  http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF
- OpenMP Specifications/Documentation:
  http://openmp.org/wp/openmp-specifications/
- OpenMP tutorial from LLNL
  https://computing.llnl.gov/tutorials/openMP/

Resources for Users: Getting Started

- About Jaguar

  http://www.nccs.gov/computing-resources/jaguar/

- Quad Core AMD Opteron Processor Overview

  http://www.nccs.gov/wp-content/uploads/2008/04/amd_craywkshp_apr2008.pdf

- PGI Compilers for XT5

  http://www.nccs.gov/wp-content/uploads/2008/04/compilers.ppt

- NCCS Training & Education – archives of NCCS workshops and seminar series, HPC/parallel computing references

  http://www.nccs.gov/user-support/training-education/

- 2009 Cray XT5 Quad-core Workshop

  http://www.nccs.gov/user-support/training-education/workshops/2008-cray-xt5-quad-core-workshop/

Go to Menu

Resources for Users: Advanced Topics

- Debugging Applications Using TotalView

  http://www.nccs.gov/user-support/general-support/software/totalview

- Using Cray Performance Tools - CrayPat

  http://www.nccs.gov/computing-resources/jaguar/debugging-optimization/cray-pat/

- I/O Tips for Cray XT4

  http://www.nccs.gov/computing-resources/jaguar/debugging-optimization/io-tips/

- NCCS Software

  http://www.nccs.gov/computing-resources/jaguar/software/

Resources for Users: More Information

- NCCS website

  http://www.nccs.gov/

- Cray Documentation

  http://docs.cray.com/

- Contact us

  help@nccs.gov