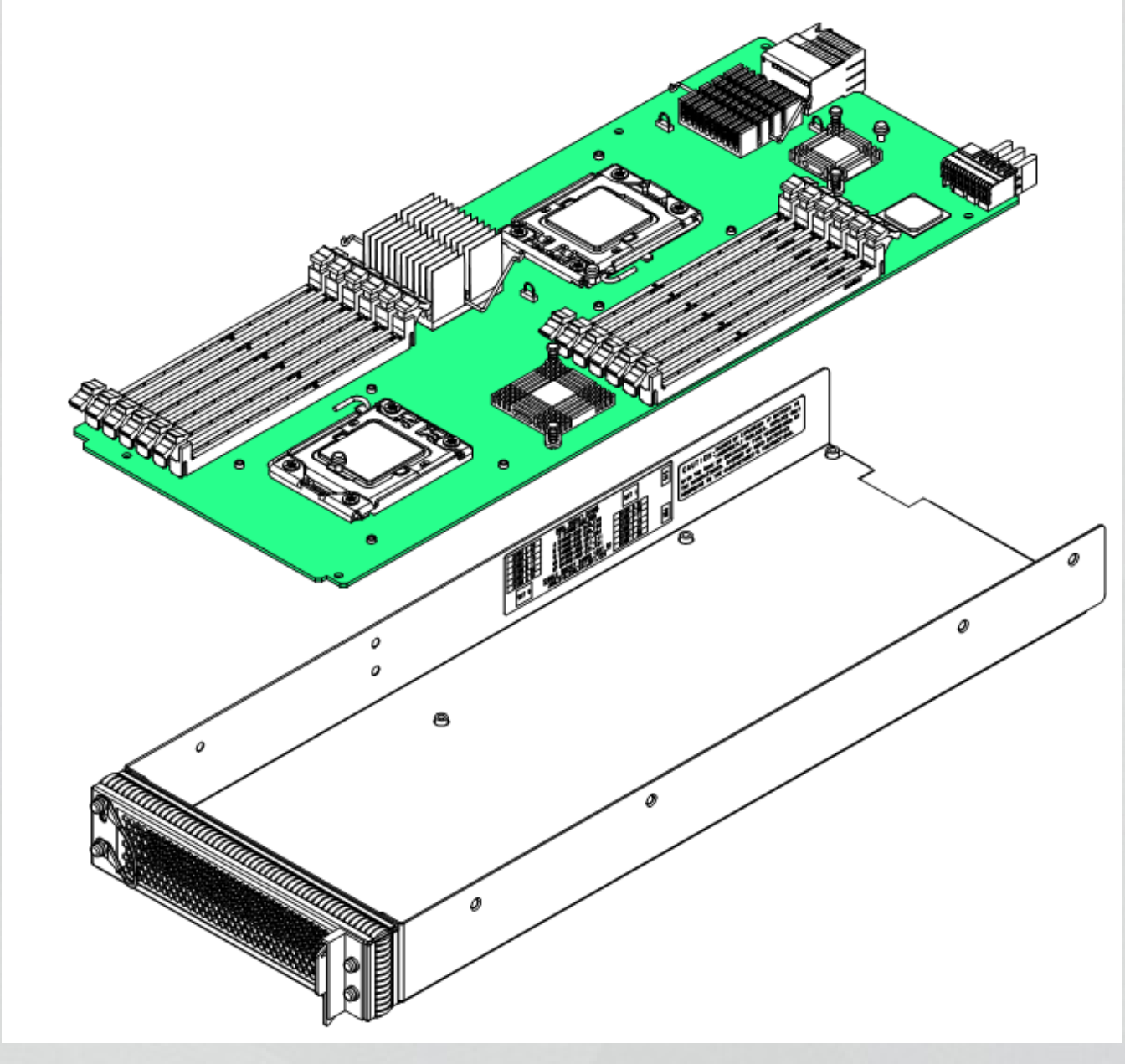



Master Class: Programming on Supercomputers

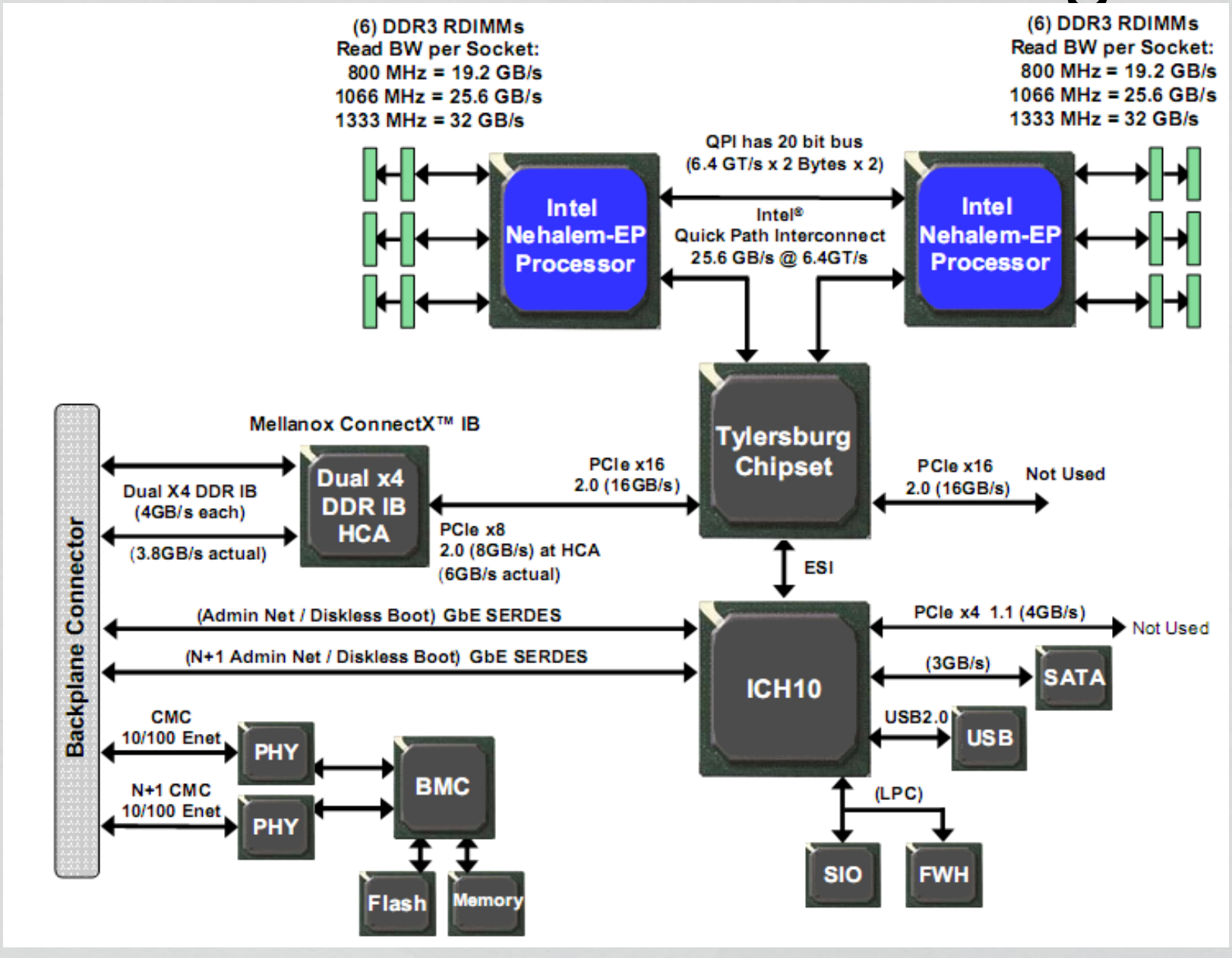
Resource – Intel X86_64 Application Dev

Intel XEON 5500 (Nehalem) Microprocessor

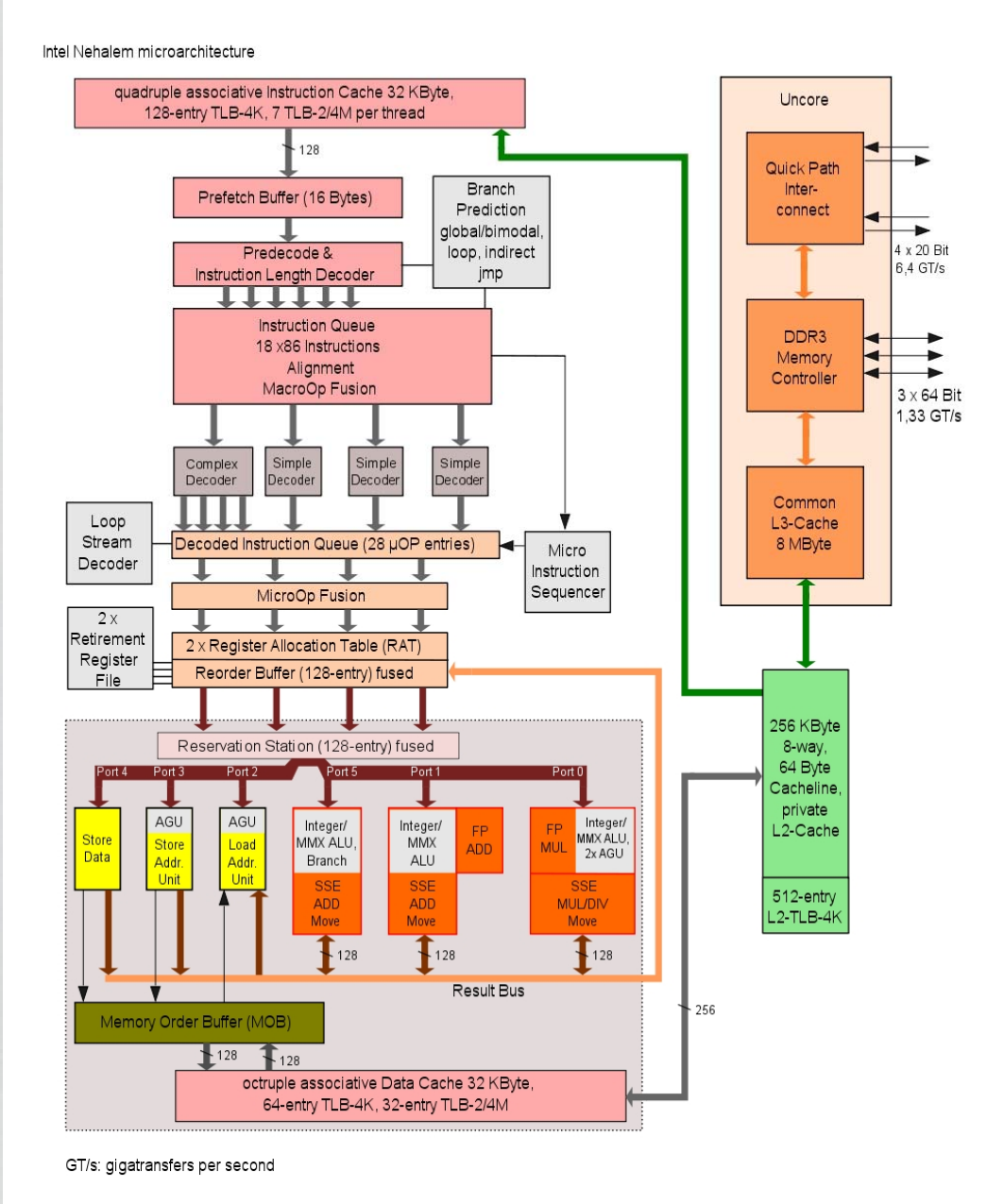
SGI ICE IP95 Blade



Nehalem node block diagram



Intel Micro-Architecture



Intel I7 Register Sets

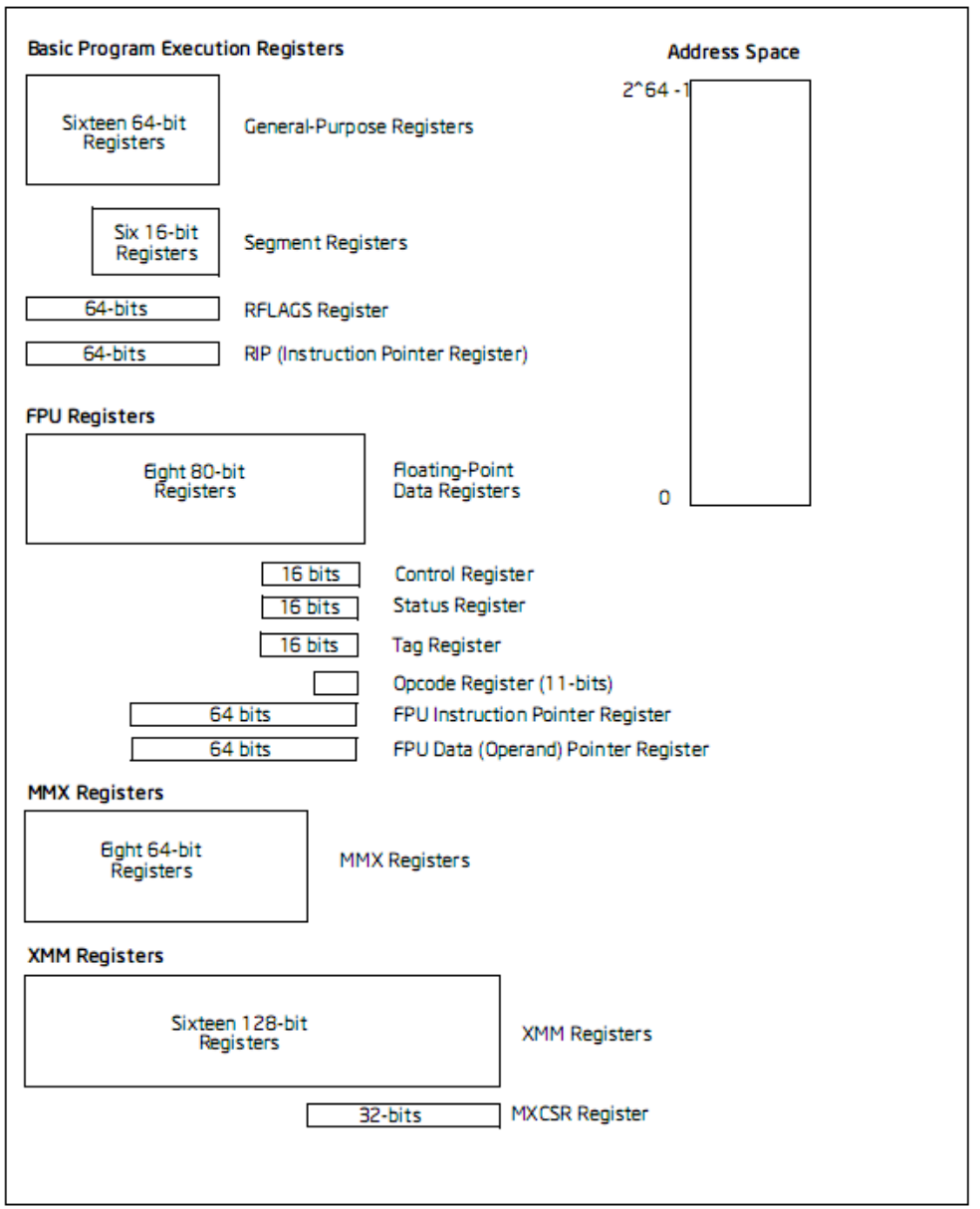


Figure 3-2. 64-Bit Mode Execution Environment

Intel I7 SIMD extension

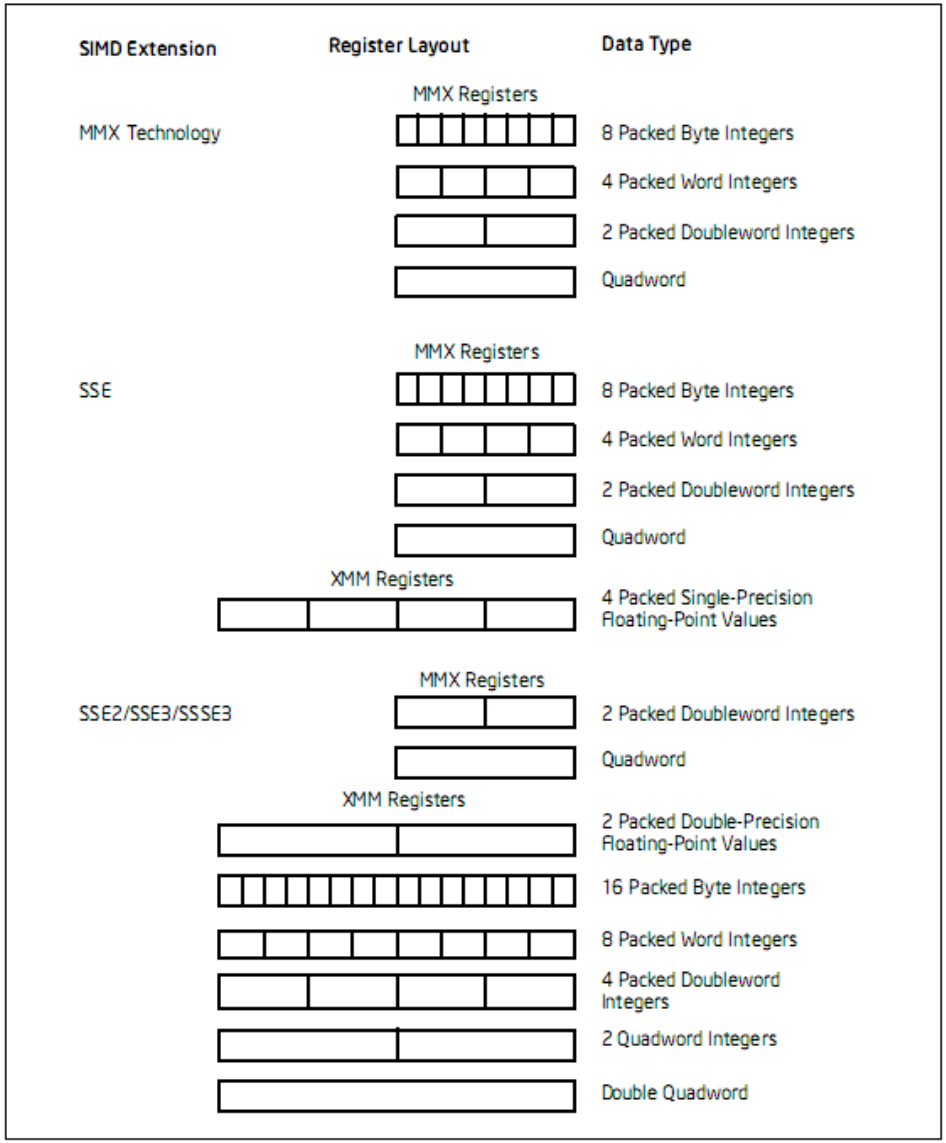
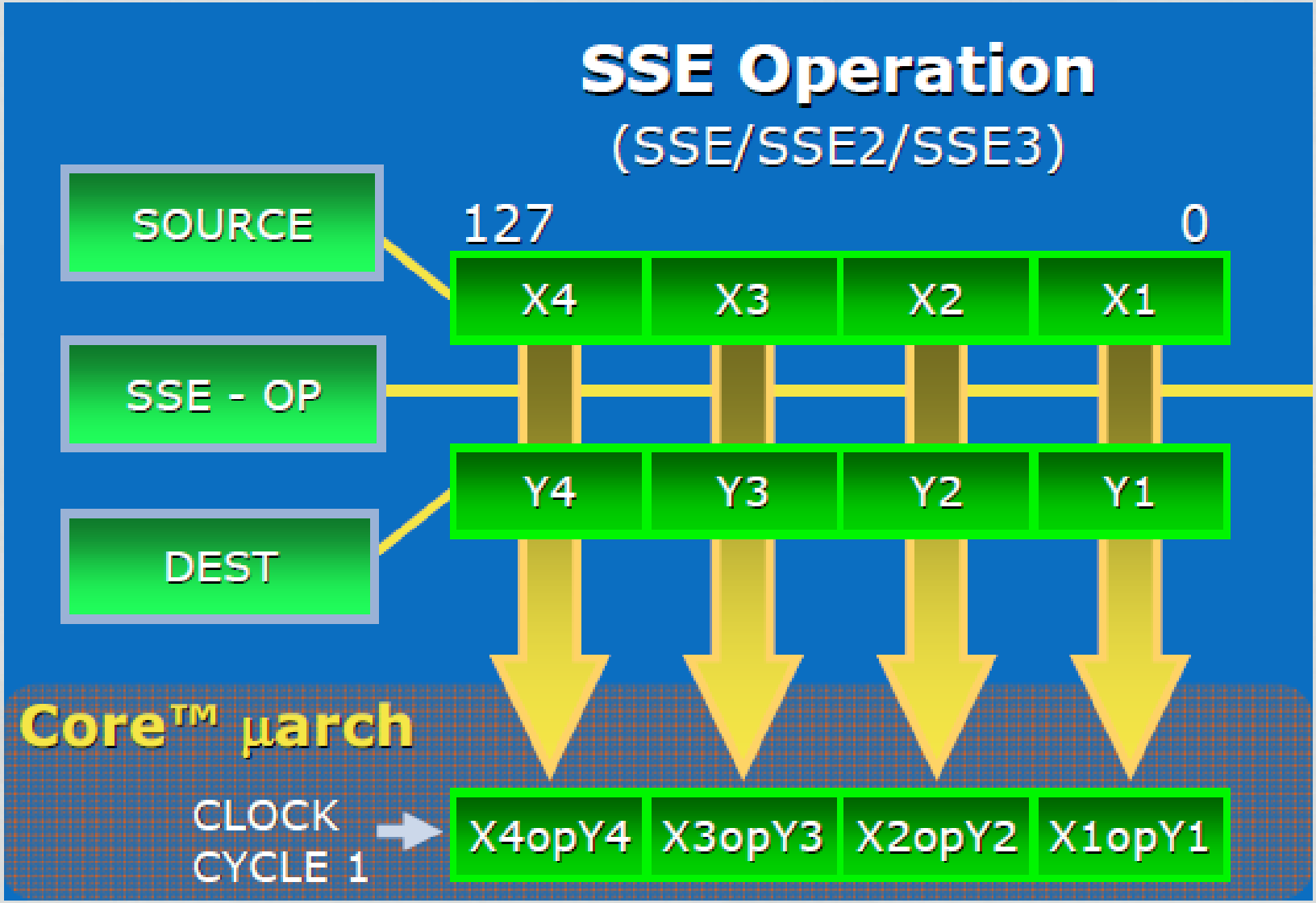
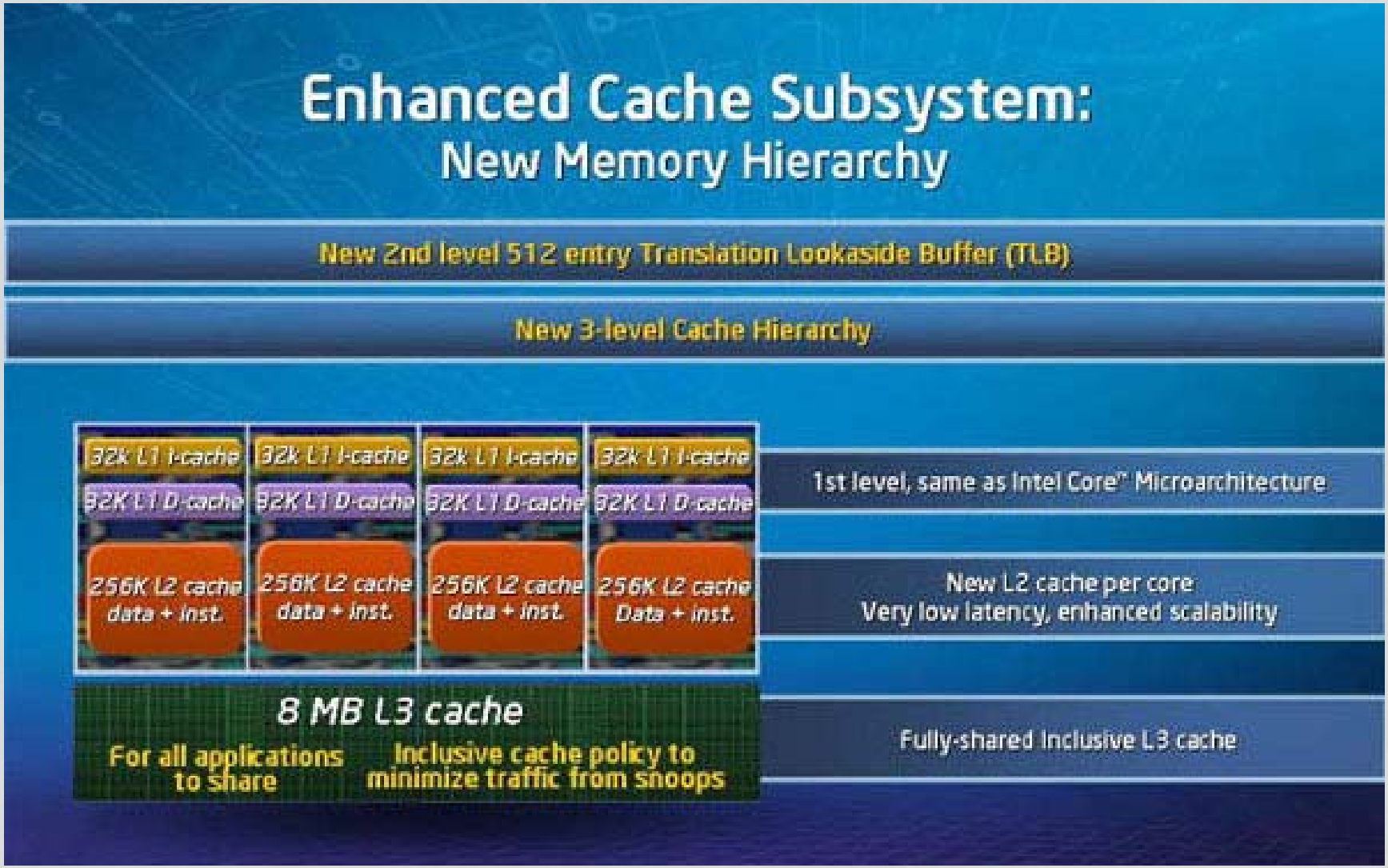


Figure 2-4. SIMD Extensions, Register Layouts, and Data Types

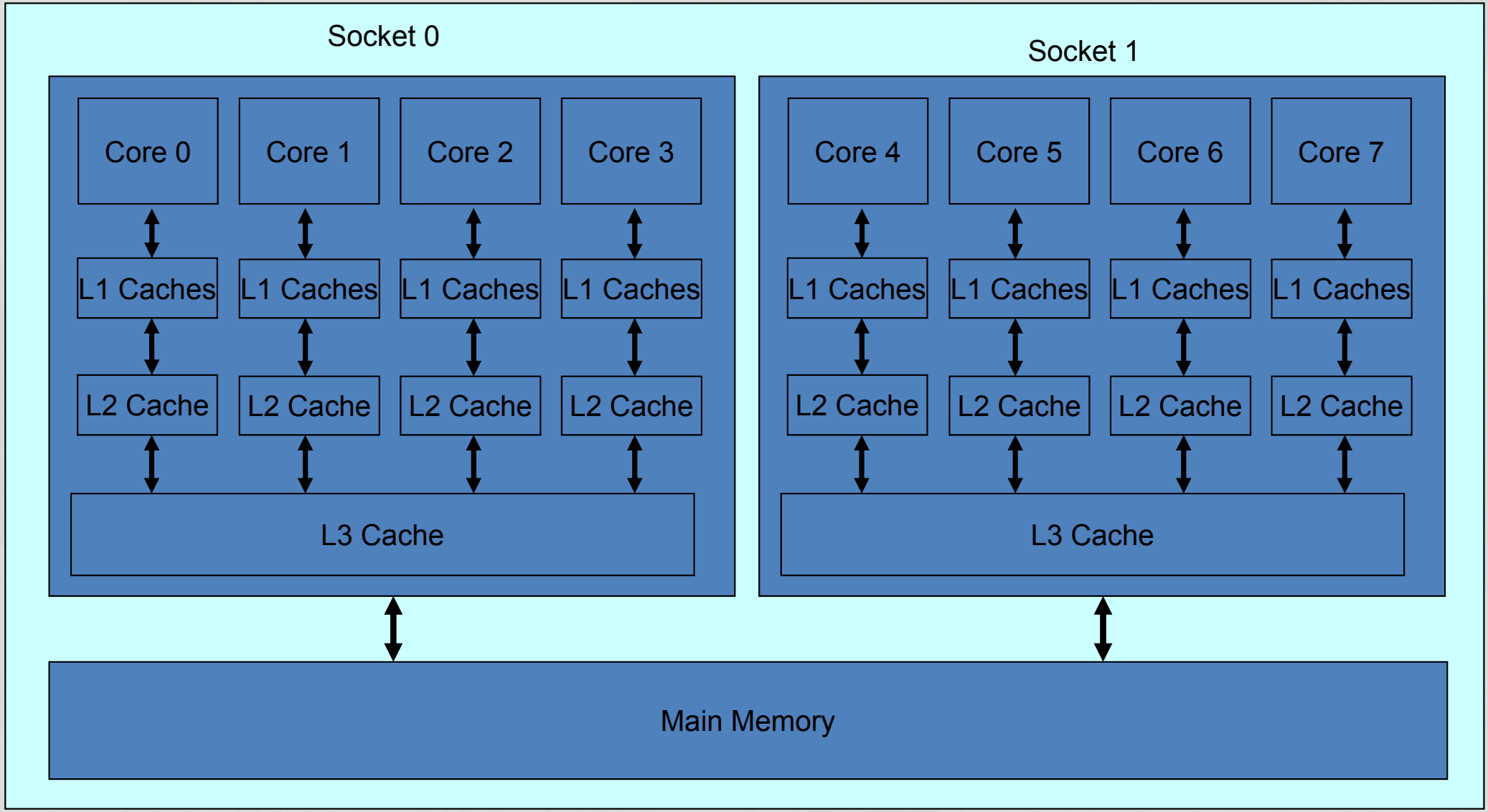
Intel XMM (Vector) Registers



Intel XEON 5500 Cache Subsystem



Nehalem node: application programming perspective



Intel C / C++ Compiler

Intel C / C++ Compiler : Environment

Steps to using the Intel C/C++ compiler:

- Set up the environment:
  - `> source <install-dir>/bin/iccvars.sh <arg>`
  - `> source <install-dir>/bin/iccvars.csh <arg>`
  - The scripts take an argument, `<arg>` specifying architecture:
    - `ia32` Compiler and libraries for IA-32 architectures only
    - `intel64` Compiler and libraries for Intel® 64 architectures only
    - `ia64` Compiler and libraries for IA-64 architectures only
  - Usually add this to your `.login`, `.profile` or `.cshrc` startup script

  OR

  `> module load <intel-cc>`
  - where modules are used, `<intel-cc>` will be site dependent
- Invoke the compiler:
  - C compiler
    `> icc`
  - C++ compiler
    `> icpc`

Intel C / C++ Compiler: Getting Started

Getting started:

- `> icc -help` for a summary of command line options
- `> man icc` man is your best friend!
- to find all the definitive documentation:

```
> which icc
/sw/sdev/intel/Compiler/11.0/074/bin/intel64/icc
ls /sw/sdev/intel/Compiler/11.0/074/Documentation
```

Compiler version information:

```
> icc –v
> icc –V
> icc --version
```

Intel C / C++ Compiler: Hello World!

The simplest case: compile AND link `test1.c` to produce executable `test1`:

```
> icc –o test1 test1.c
```

The following separates the compile and link (ld) phases:

```
> icc –c test1.c          compiles an object file: test1.o
> icc –o test1 test1.o    builds an executable file: test1
```

Save compiler version information in the executable:

```
 > icc –sox –o test1 test1.c
 > strings -a test1 |grep comment
-?comment:Intel(R) C++ Compiler Professional for applications running
   on Intel(R) 64, Version 11.0    Build 20081105 %s  : test1.c : -sox
   -o test1 -?comment:Intel(R) C++ Compiler Professional for
   applications running on Intel(R) 64, Version 11.0    Build 20081105
   %s  : test1.c : -sox -o test1 -defaultlib:libirc -defaultlib:libirc
```

Intel C / C++ Compiler: Default Behaviour

If you do not specify any options when you invoke the Intel® C/C++ Compiler, the compiler performs the following:

- produces an executable file (`a.out` by default)
- invokes options specified in a configuration file first
- invokes options specified on the command line
- searches for header files in known locations
- sets 16 bytes as the strictest alignment constraint for structures
- displays error and warning messages
- uses ANSI with extensions
- performs standard optimizations (usually `-O2`)
- on operating systems that support characters in Unicode* (multi-byte) format, the compiler will process file names containing these characters

Intel C / C++ Compiler: Input File Processing

Input File Name	Interpretation	Action
file.c	C source file	Passed to compiler
file.C, file.CC, file.cc, file.cpp, file.cxx	C++ source file	Passed to compiler
file.a, file.so	Library file	Passed to linker
file.i	Preprocessed file	Passed to stdout
file.o	Object file	Passed to linker
file.s	Assembly file	Passed to assembler

Intel C / C++ Compiler: Output Files Produced

Compiler output files:

file.i Preprocessed file -- produced with the `-P` option.

file.o Object file -- produced with the `-c` option.

file.s Assembly language file -- produced with the `-S` option.

a.out Executable file -- produced by default compilation

Intel C / C++ Compiler: Debugging

Debugging:

```
> icc –g test1 test1.c
```

Generates code to support symbolic debugging.

Warning: This option changes the default optimization from `–O2` to `–O0`
That is: **no optimisation**!

If this is not what you want, then use:

```
> icc –g –O2 test1 test1.c
```

The compiler supports `gdb`, `idb` (gui version), and `idbc` (Command lLne Version).

Intel C / C++ Compiler: Stack Traceback

- `traceback` tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.

The default (`-notraceback`) is to produce no traceback information.

Intel C / C++ Compiler: Warnings

Use the following compiler options to control remarks, warnings, and errors:

Option	Result
`-w`	Suppress all warnings
`-w0`	Display only errors
`-w1`	Display only errors and warnings (**default**)
`-w2`	Display errors, warnings, and remarks
`-Wbrief`	Display brief one-line diagnostics
`-Wcheck`	Enable more strict diagnostics
`-Werror-all`	Change all warnings and remarks to errors
`-Werror`	Change all warnings to errors

Intel C / C++ Compiler: Target Processor

Ensure that you are compiling for the correct target system.

This is especially important on cluster systems because the login-in (service) nodes may not have the same CPU as the compute nodes.

`-x[processor]` specifies the processor for which code is generated.

Possible values are:

`-xhost`	
`-xsse2`	
`-xsse3`	
`-xssse3`	Clovertown
`-xsse4.1`	Harpertown
`-xsse4.2`	i5, i7, Nehalem

Intel C / C++ Compiler: linux cpuinfo

```
> cat /proc/cpuinfo
processor       : 3
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Xeon(R) CPU           X5350  @ 2.66GHz
stepping        : 7
cpu MHz         : 1600.000
cache size      : 4096 KB
physical id     : 1
siblings        : 4
core id         : 1
cpu cores       : 4
fpu       : yes
fpu_exception   : yes
cpuid level     : 10
wp        : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
    dts acpi mmx fxsr sse sse2 ss ht tm syscall nx lm constant_tsc pni monitor ds_cpl vmx est tm2
    ssse3 cx16 xtpr dca lahf_lm
bogomips        : 5333.55
clflush size    : 64
cache_alignment     : 64
address sizes   : 36 bits physical, 48 bits virtual
power management    :
```

Intel C/C++ Compiler: floating point options

`-fp-model`	Specifies semantics used in floating-point calculations. Values are `precise`, `fast` [`=1/2`], `strict`, `source`, `double`, `extended`, [`no-`]`except`.
`-fp-speculation`	Specifies the speculation mode for floating-point operations. Values are `fast`, `safe`, `strict`, and `off`.
`-prec-div`	Attempts to use slower but more accurate implementation of floating-point divide. Use this option to disable the divide optimizations in cases where it is important to maintain the full range and precision for floating-point division. Using this option results in greater accuracy with some loss of performance. Specifying `-no-prec-div` enables optimizations that result in slightly less precise results than full IEEE division.
`-complex-limited-range`	Enables the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX. This can cause performance improvements in programs that use a lot of COMPLEX arithmetic. Values at the extremes of the exponent range might not compute correctly.
`-ftz`	This option only has an effect when the main program is being compiled. This option flushes denormal results to zero when the application is in the gradual underflow mode. It may improve performance if denormal values are not critical to your application's behavior.

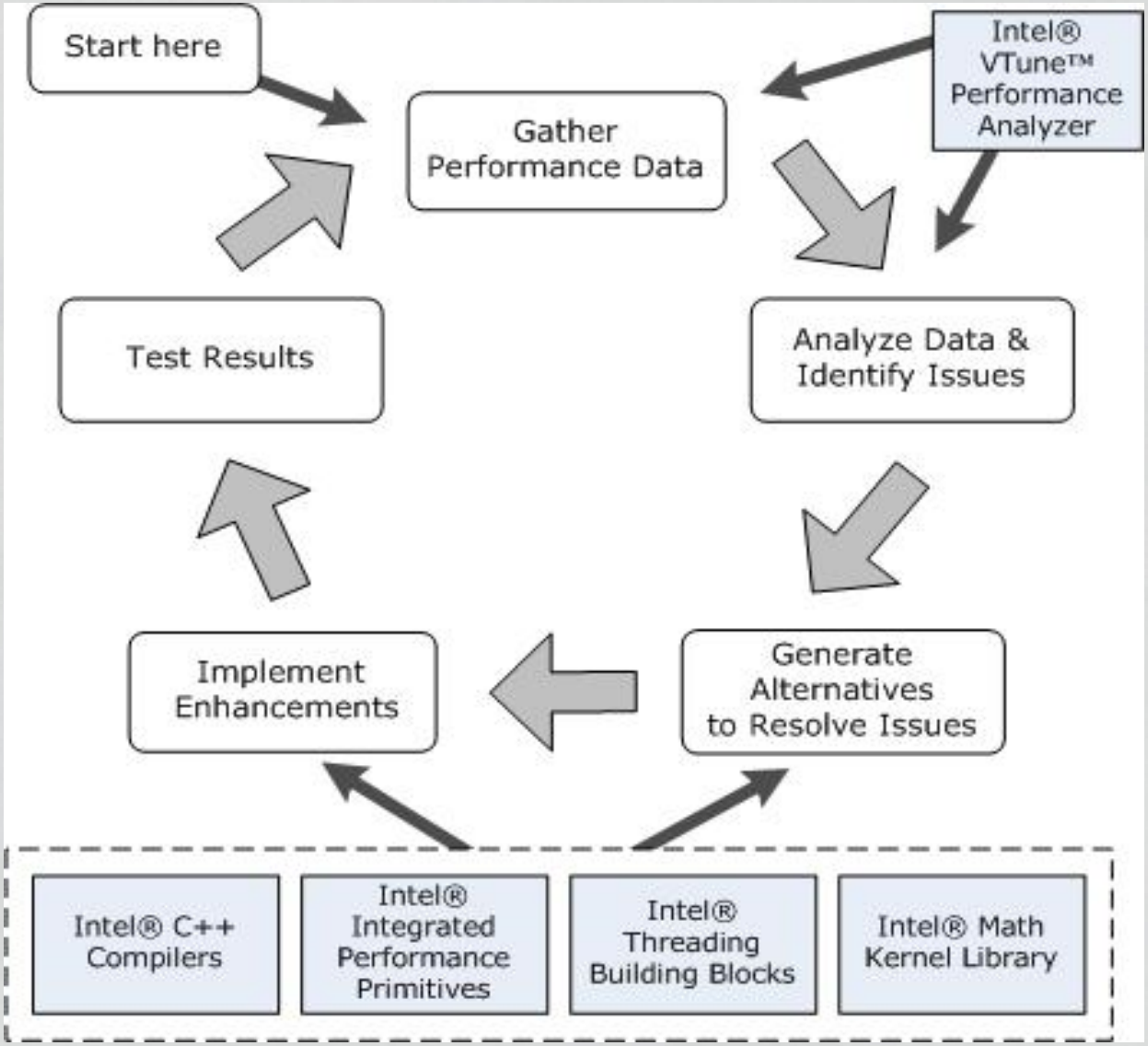
Intel C/C++ Compiler: floating point options (continued)

`-prec-sqrt`	Improves the accuracy of square root implementations, but using this option may impact speed.
`-pc`	Changes the floating point significand precision. Use this option when compiling applications. The application must use `main()` as the entry point, and you must compile the source file containing `main()` with this option.
`-rcd`	Disables rounding mode changes for floating-point-to-integer conversions
`-fp-port`	Causes floating-point values to be rounded to the source precision at assignments and casts.
`-mp1`	This option rounds floating-point values to the precision specified in the source program prior to comparisons. It also implies `-prec-div` and `-prec-sqrt`. This option has less impact to performance and disables fewer optimizations than the `-fp-model` precise option.

Intel C/C++ Compiler: -fp-model options

`precise`	Enables value-safe optimizations on floating-point data.
`fast=1,fast=2`	Enables more aggressive optimizations on floating-point data.
`strict`	Enables `precise` and `except`, disables contractions, and enables pragma stdc fenv_access.
`source`	Rounds intermediate results to source-defined precision and enables value-safe optimizations.
`double`	Rounds intermediate results to 53-bit (double) precision.
`extended`	Rounds intermediate results to 64-bit (extended) precision .
`[no-]except`	Determines whether floating-point exception semantics are used.

Intel optimization flow chart



Intel C / C++ Compiler: Optimization

Option	Intel	gcc
`-O0`	Turns off optimization.	**Default**. Turns off optimization. Same as -O.
`-O1`	Decreases code size with some increase in speed	Decreases code size with some increase in speed.
`-O2` `-O`	**Default**. Favors speed optimization with some increase in code size. Same as -O. Intrinsics, loop unrolling, and inlining are performed.	Optimizes for speed as long as there is not an increase in code size. Loop unrolling and function inlining, for example, are not performed
`-O3`	Enables -O2 optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop and memory access transformations	Optimizes for speed while generating larger code size. Includes -O2 optimizations plus loop unrolling and inlining. Similar to -O2 -ip on Intel compiler.
`-fast`	Enables a collection of common, recommended optimizations for run-time performance. Can introduce architecture dependency	

Intel C/C++ Compiler: assembly listing

```
double add_2(double a, double b) {
   return a + b;
}
```

The –S option compiles to an assembly file, `add_2.s` containing the assembler code

```
> icc –S add_2.c
```

```
add_2:
# parameter 1: %xmm0
# parameter 2: %xmm1
..B1.1:                         # Preds ..B1.0
..___tag_value_add_2.1:                                         #1.34
        addsd     %xmm1, %xmm0                                  #2.15
        ret                                                     #2.15
```

```
> icc –c add_2.s
```

Will create a `add_2.o` file.

Intel C/C++ Compiler: automatic vectorization overview

The automatic vectorizer (also called the auto-vectorizer) is a component of the Intel® compiler that automatically uses SIMD instructions in the MMX™, Intel® Streaming SIMD Extensions (Intel® SSE, SSE2, SSE3 and SSE4 Vectorizing Compiler and Media Accelerators) and Supplemental Streaming SIMD Extensions (SSSE3) instruction sets.

The vectorizer detects operations in the program that can be done in parallel, and then converts a number of sequential operations to one SIMD instruction that processes 2, 4, 8 or 16 elements in parallel, depending on the data type.

Intel C/C++ Compiler: automatic vectorization overview

- The compiler supports a variety of directives that can help the compiler to generate effective vector instructions.
  - automatic vectorization
  - vectorization with user intervention
- Vectorization options:

`-x`	Generates specialized code to run exclusively on processors with the extensions specified as the *processor* value. See *Targeting IA-32 and Intel® 64 Architecture Processors Automatically* for more information about using the option.
`-ax`	Generates, in a single binary, code specialized to the extensions specified as the *processor* value and also generic IA-32 architecture code. The generic code is usually slower. See *Targeting Multiple IA-32 and Intel® 64 Architecture Processors for Run-time Performance* for more information about using the option
`-vec` `-no-vec`	Enables or disables vectorization and transformations enabled for vectorization. The default is that vectorization is enabled.
`-vec-report`	Controls the diagnostic messages from the vectorizer.

Intel C/C++ Compiler: vectorization reports

`-vec-report[n]`

`0`	Tells the vectorizer to report no diagnostic information.
`1`	Tells the vectorizer to report on vectorized loops. (**default**)
`2`	Tells the vectorizer to report on vectorized and non-vectorized loops.
`3`	Tells the vectorizer to report on vectorized and non-vectorized loops and any proven or assumed data dependences.
`4`	Tells the vectorizer to report on non-vectorized loops.
`5`	Tells the vectorizer to report on non-vectorized loops and the reason why they were not vectorized.

Intel C/C++ Compiler: guidelines for vectorization

Programming Guidelines for Vectorization

- The goal of vectorizing compilers is to exploit single-instruction multiple data (SIMD) processing automatically. Users can help, however, by supplying the compiler with additional information; for example, by using directives.

Guidelines

- You will often need to make some changes to your loops. Follow these guidelines for loop bodies.

Use:

- straight-line code (a single basic block)
- vector data only; that is, arrays and invariant expressions on the right hand side of assignments. Array references can appear on the left hand side of assignments.
- only assignment statements

Intel C/C++ Compiler: tips for vectorization

Avoid:

- function calls
- unvectorizable operations (other than mathematical)
- mixing vectorizable types in the same loop
- data-dependent loop exit conditions

To make your code vectorizable, you will often need to make some changes to your loops. However, you should make only the changes needed to enable vectorization and no others. In particular, you should avoid these common changes:

- loop unrolling; the compiler does it automatically.
- decomposing one loop with several statements in the body into several single-statement loops.

Intel C/C++ Compiler: restrictions for vectorization

Restrictions:

There are a number of restrictions that you should consider. Vectorization depends on two major factors:

- **Hardware**

  The compiler is limited by restrictions imposed by the underlying hardware. In the case of Streaming SIMD Extensions, the vector memory operations are limited to stride-1 accesses with a preference to 16-byte-aligned memory references. This means that if the compiler abstractly recognizes a loop as vectorizable, it still might not vectorize it for a distinct target architecture

- **Style of source code**

  The style in which you write source code can inhibit optimization. For example, a common problem with global pointers is that they often prevent the compiler from being able to prove that two memory references refer to distinct locations. Consequently, this prevents certain reordering transformations.

Intel C/C++ Compiler: Interprocedural Optimization (IPO)

InterProcedural Optimization is an automatic, multi-step process: compilation and linking; however, IPO supports two compilation models:

- `-ip` single-file compilation and
- `-ipo` multi-file compilation.

Intel C/C++ Compiler: Interprocedural Optimization (IPO)

- Single-file compilation results in one real object file for each source file being compiled. During single-file compilation the compiler performs inline function expansion for calls to procedures defined within the current source file.

- The compiler performs some single-file interprocedural optimization at the default optimization level, `-O2`, additionally the compiler performs some inlining for the `-O1` optimization level, like inlining functions marked with inlining pragmas or attributes (GNU C and C++) and C++ class member functions with bodies included in the class declaration.

Intel C/C++ Compiler: Interprocedural Optimization (IPO)

`-ipo`

Multi-file compilation results in one or more mock object files rather than normal object files.

Additionally, the compiler collects information from the individual source files that make up the program.

Using this information, the compiler performs optimizations across functions and procedures in different source files.

Inlining is the most powerful optimization supported by IPO.

```
icc -opt-report n -opt-report-phase=ipo
```

Where `n`, if specified, can be `0`, `1`, `2`, or `3`. Otherwise the default is `2`.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

Profile-Guided Optimization (PGO)

- PGO improves application performance by:
  - Reorganizing code layout to reduce instruction-cache problems,
  - Shrinking code size, and
  - reducing branch mispredictions.

- PGO provides information to the compiler about the areas in an application that are most frequently executed.

- By knowing these areas, the compiler is able to be more selective and specific in optimizing the application.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

PGO consists of three phases or steps:

Step one: Instrument the program.

In this phase, the compiler creates and links an instrumented program from your source code with special code from the compiler.

Step two: Run the instrumented executable.

Each time you execute the instrumented code, the instrumented program generates a dynamic information file, which is used in the final compilation.

Step three: Final compilation.

When you compile the second time, the dynamic information files are merged into a summary file. Using the summary of the profile information in this file, the compiler attempts to optimize the execution of the most heavily traveled paths in the program.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

PGO enables the compiler to take better advantage of the processor architecture, more effective use of instruction paging and cache memory, and make better branch predictions. PGO provides the following benefits:

- Use profile information for register allocation to optimize the location of spill code.
- Improve branch prediction for indirect function calls by identifying the most likely targets. (Some processors have longer pipelines, which improves branch prediction and translates into high performance gains.)
- Detect and do not vectorize loops that execute only a small number of iterations, reducing the run time overhead that vectorization might otherwise add.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

InterProcedural Optimization (IPO) and PGO can affect each other.

Using PGO can often enable the compiler to make better decisions about function inlining, which increases the effectiveness of interprocedural optimizations.

Unlike other optimizations, such as those strictly for size or speed, the results of IPO and PGO vary. This variability is due to the unique characteristics of each program, which often include different profiles and different opportunities for optimizations.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

Performance Improvements with PGO

PGO works best for code with many frequently executed branches that are difficult to predict at compile time.

An example is a code with intensive error-checking in which the error conditions are false most of the time.

The infrequently executed (cold) error-handling code can be relocated so the branch is rarely predicted incorrectly.

Minimizing cold code interleaved into the frequently executed (hot) code improves instruction cache behavior.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

When you use PGO, consider the following guidelines:

- Minimize changes to your program after you execute the instrumented code and before feedback compilation.
  During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated.
  If you modify your program, the compiler issues a warning that the dynamic information does not correspond to a modified function.

- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

- Know the sections of your code that are the most heavily used.
  If the data set provided to your program is very consistent and displays similar behavior on every execution, then PGO can probably help optimize your program execution.

- Different data sets can result in different algorithms being called.
  The difference can cause the behavior of your program to vary for each execution. In cases where your code behavior differs greatly between executions, PGO may not provide noticeable benefits.
  If it takes multiple data sets to accurately characterize application performance then execute the application with each of these data sets then merge the dynamic profiles; this technique should result in an optimized application.

You must insure the benefit of the profiled information is worth the effort required to maintain up-to-date profiles.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

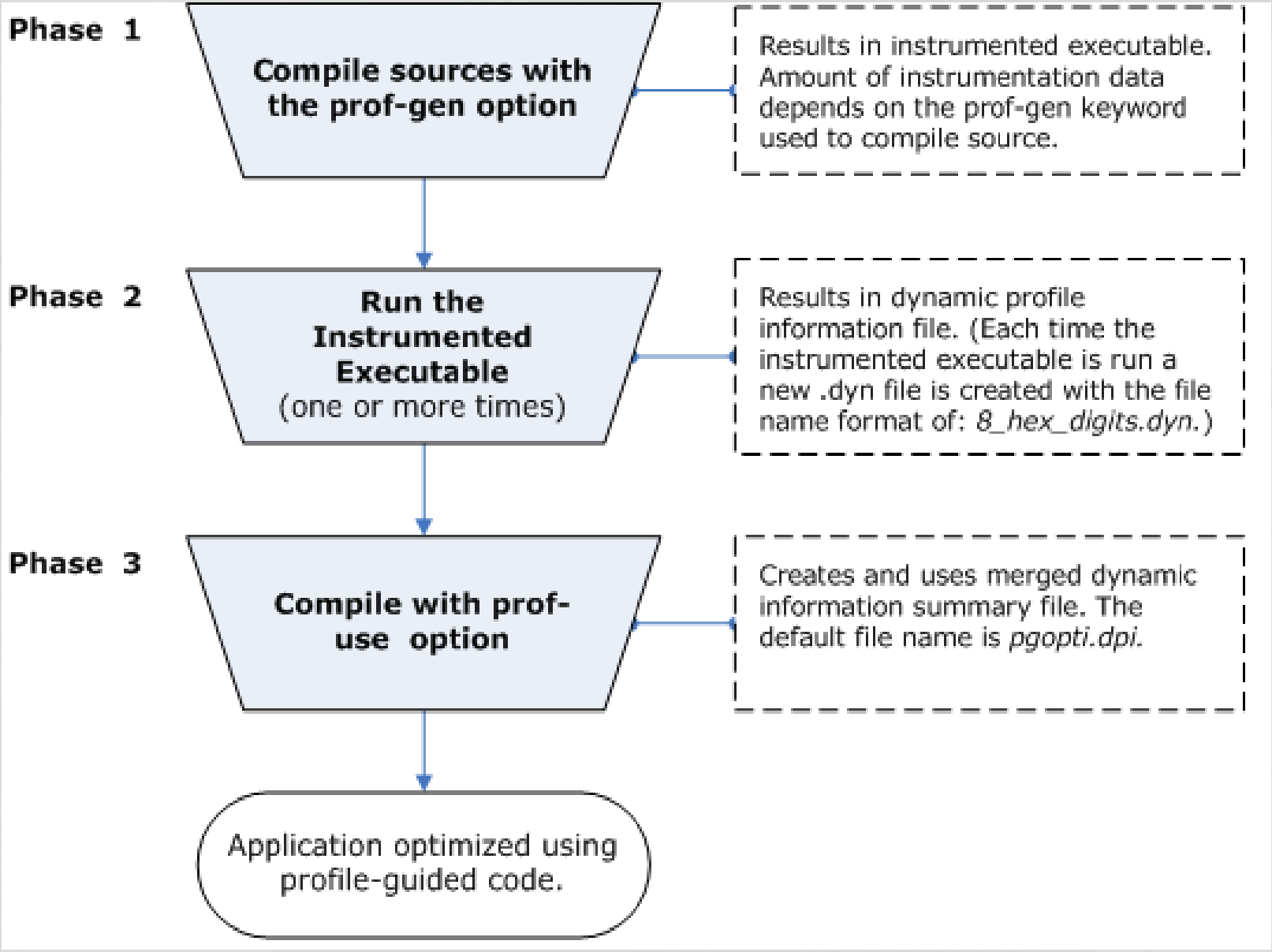
PGO consists of three phases (or steps):

1. Generating instrumented code by compiling with the `-prof-gen` option when creating the instrumented executable.

2. Running the instrumented executable, which produces dynamic-information (`.dyn`) files.

3. Compiling the application using the profile information using the `-prof-use` option.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)



Intel C/C++ Compiler: Code Coverage Tool

The code coverage tool, codecov, provides software developers with a view of how much application code is exercised when a specific workload is applied to the application.

To determine which code is used, the code coverage tool uses Profile-guided Optimization (PGO) options and optimizations.

The major features of the code coverage tool are:

- Visually presenting code coverage information for an application with a customizable code coverage coloring scheme
- Displaying dynamic execution counts of each basic block of the application
- Providing differential coverage, or comparison, profile data for two runs of an application

Intel C/C++ Compiler: Code Coverage Tool

To use `codecov` on an application, the following items must be available:

- The application sources.
- The `.spi` file generated by the compiler when compiling the application for the instrumented binaries using the `-prof-gen=srcpos` options.
- A `pgopti.dpi` file that contains the results of merging the dynamic profile information (`.dyn`) files, which is most easily generated by the `profmerge` tool.
  This file is also generated implicitly by the Intel® compilers when compiling an application with `-prof-use` options with available `.dyn` and `.dpi` files.

Intel C/C++ Compiler: Code Coverage Tool

CODE_COVERAGE.HTML

Intel C/C++ Compiler: High-Level Optimization (HLO)

High-Level Optimization (HLO)

HLO exploits the properties of source code constructs (for example, loops and arrays) in applications developed in high-level programming languages. Within HLO, loop transformation techniques include:

- Loop Permutation or Interchange
- Loop Distribution
- Loop Fusion
- Loop Unrolling
- Data Prefetching
- Scalar Replacement
- Unroll and Jam
- Loop Blocking or Tiling
- Partial-Sum Optimization

Intel C/C++ Compiler: High-Level Optimization (HLO)

- Loadpair Optimization
- Predicate Optimization
- Loop Versioning with Low Trip-Count Check
- Loop Reversal
- Profile-Guided Loop Unrolling
- Loop Peeling
- Data Transformation: Malloc Combining and Memset Combining
- Loop Rerolling
- Memset and Memcpy Recognition
- Statement Sinking for Creating Perfect Loopnests

Intel C/C++ Compiler: High-Level Optimization (HLO)

Some HLO is done with default optimization, `-O2`

However, most HLO is done at optimization level, `-O3`

`-unroll`*n* Specifies the maximum number of times a loop is unrolled.

`-unroll0` Disables loop unrolling

```
#pragma unroll
#pragma unroll(n)
#pragma nounroll
```

Indicates to the compiler to unroll or not to unroll a counted loop. These pragmas are supported only with `-O3`.

Intel C/C++ Compiler: High-Level Optimization (HLO)

When loops are dependent, improving loop performance becomes much more difficult. Loops can be dependent in several, general ways:

- **Flow Dependency**, read after write:

```
for (int j=1; j<1024; j++)
  A[j]=A[j-1];
```

- **Anti Dependency**, write after read:

```
for (int j=1; j<1024; j++)
  A[j]=A[j+1];
```

- **Output Dependency**, write after write:

```
for (int j=1; j<1024; j++) {
   A[j]=B[j];
   A[j+1]=C[j];
}
```

Intel C/C++ Compiler: High-Level Optimization (HLO)

Reductions: The Intel® compiler can successfully vectorize most loops containing reductions on simple math operators like multiplication (*), addition (+), subtraction (-), and division (/). Reductions collapse array data to scalar data by using associative operations:

```
sum = 0.0;
for (int j=0; j<MAX; j++) {
    sum += c[j];
}
```

Intel C/C++ Compiler: Automatic Parallelization

The auto-parallelization feature of the Intel® compiler automatically translates serial portions of the input program into equivalent multi-threaded code.

The auto-parallelizer analyzes the dataflow of the loops in the application source code and generates multi-threaded code for those loops which can safely and efficiently be executed in parallel.

This allows applications to exploit the full potential of the parallel architecture found in current symmetric multi-processor (SMP) systems.

The parallel run-time support provides the same run-time features as found in OpenMP, such as handling the details of loop iteration modification, thread scheduling, and synchronization.

Intel C/C++ Compiler: Automatic Parallelization

`-parallel`	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. Depending on the program and level of parallelization desired, you might need to set the `KMP_STACKSIZE` environment variable to a larger size.
`-par-threshold[n]`	Sets a threshold for the auto of loops based on the probability of profitable execution of the loop in parallel. Valid values of n can be 0 to 100. (default is 100)
`-par-schedule-`*`keyword[=n]`*	Specifies the scheduling algorithm or a tuning method for loop iterations. It specifies how iterations are to be divided among the threads of the team.
`-par-report[n]`	Controls the diagnostic levels in the auto-parallelizer optimizer. Valid values of n can be 0, 1, 2, 3 (default is 1)

Intel C/C++ Compiler: Automatic Parallelization

`-par-schedule-keyword[=n]`

`auto`	Lets the compiler or run-time system determine the scheduling algorithm .
`static`	Divides iterations into contiguous pieces.
`static-balanced`	Divides iterations into even-sized chunks.
`static-steal`	Divides iterations into even-sized chunks, but allows threads to steal parts of chunks from neighboring threads.
`dynamic`	Gets a set of iterations dynamically.
`guided`	Specifies a minimum number of iterations.
`guided-analytical`	Divides iterations by using exponential distribution or dynamic distribution.
`runtime`	Defers the scheduling decision until run time.

Intel C/C++ Compiler: Automatic Parallelization

Three requirements must be met for the compiler to automatically parallelize a loop:

1. The number of iterations must be known before entry into a loop so that the work can be divided in advance. A while-loop, for example, cannot usually be parallelized.
2. There can be no jumps into or out of the loop.
3. The loop iterations must be independent.

If you know that parallelizing a particular loop is safe and that potential aliases can be ignored, you can instruct the compiler to parallelize the loop using:
`#pragma parallel`

Intel C/C++ Compiler: Parallel Environment Variables

Auto-parallelization uses the following OpenMP and KMP environment variables:

`OMP_NUM_THREADS`	Sets the maximum number of threads to use for parallel regions.
`KMP_STACKSIZE`	Sets the number of bytes to allocate for each OpenMP* thread to use as the private stack for the thread. Use the optional suffixes: b (bytes), k (kilobytes), m (megabytes), g (gigabytes), or t (terabytes) to specify the units.
`OMP_SCHEDULE`	Sets the run-time schedule type and an optional chunk size.
`KMP_VERSION`	Enables (1) or disables (0) the printing of OpenMP run-time library version information during program execution. Default is 0.

Intel C/C++ Compiler: Optimization Reports

`-opt-report[n]`

`0`	Tells the compiler to generate no optimization report.
`1`	Tells the compiler to generate a report with the minimum level of detail.
`2`	Tells the compiler to generate a report with the medium level of detail. This is the default if n is not specified.
`3`	Tells the compiler to generate a report with the maximum level of detail.

Intel C/C++ Compiler: Optimization Reports

`-opt-report-phase=`

`ipo`	The Interprocedural Optimizer phase
`hlo`	The High Level Optimizer phase
`hpo`	The High Performance Optimizer phase
`ilo`	The Intermediate Language Scalar Optimizer phase
`pgo`	The Profile Guided Optimization phase
`all`	All optimizer phases

The default is to produce no optimization reports.

Intel Fortran Compiler

Intel Fortran Compiler:

The Intel Fortran Compiler Version 11.0.074 supports:

- Fortran 90
- Fortran 95
- many Fortran 2003 language features
- OpenMP 3.0
- Cluster OpenMP

Intel Fortran Compiler: Environment

Steps to using the Intel Fortran compiler:

- Set up the environment:
  - `> source <install-dir>/bin/ifortvars.sh <arg>`
  - `> source <install-dir>/bin/ifortvars.csh <arg>`
  - The scripts take an argument specifying architecture:
    - ia32: Compiler and libraries for IA-32 architectures only
    - intel64: Compiler and libraries for Intel® 64 architectures only
    - ia64: Compiler and libraries for IA-64 architectures only
  - Usually add this to the `.login`, `.profile` or `.cshrc` startup script OR
  - `> module load <intel-fortran>`
  - where modules are used, `<intel-fortran>` will be site dependent
- Invoke the compiler:
  - `> ifort [options] input_files`

Intel Fortran Compiler: Compilation Phases

The `ifort` command is really a driver that calls the following components as required:

Phase	**Software**
Preprocess	`fpp`
Compile	`fortcom`
Assemble	`as`
Link	`ld`

Intel Fortran Compiler: Default Behaviour

By default, `ifort`, the compiler driver generates executable file(s) from input file(s) and performs the following actions:

- Displays certain diagnostic messages, warnings, and errors.
- Performs default settings and optimizations, unless these options are overridden by specific options settings.
- Searches for source files in the current directory or in the directory path explicitly specified before a file name (for example, looks in `src` when the directory path is `src/test.f90`).
- Searches for include and module files in:
  - The directory path explicitly specified before a file name (for example, looks in `src` when the including source is specified as `src/test.f90`)
  - **The current directory**
  - The directory specified by using the `-module` path option (for all module files)
  - The directory specified by using the `-I`*`dir`* option (for module files referenced in USE statements and include files referenced in `INCLUDE` statements.)
  - Any directory explicitly specified in any `INCLUDE` within an included file
- Passes options designated for linking as well as user-defined libraries to the linker. The linker searches for any library files in directories specified by the `LIB` variable, if they are not found in the current directory.

Intel Fortran Compiler: The Best News!

The Intel C/C++ and Fortran Compilers share many features.

Almost everything described in the previous C/C++ module applies to Fortran codes.

Intel Fortran Compiler: Input Files

Passed to the compiler, `fortcom`:

Fortran fixed-form source: `fred.f fred.for fred.ftn fred.i`
Fortran free-form source: `fred.f90 fred.i90`

Passed to the preprocessor, `fpp`, and then to the compiler `fortcom`:

Fortran fixed-form source: `fred.F fred.FOR fred.FTN fred.FPP fred.fpp`
Fortran free-form source: `fred.F90`

Passed to the assembler, `as`: `fred.s`

Passed to the linker, `ld`:

object library: `fred.a`
object file: `fred.o`

Intel Fortran Compiler: Output Files

`ifort` can produce the following output files:

- An executable file, `a.out` by default, if you omit the `–o <filename>` option.
- An object file, if you specify the `-c` option on the command line. An object file is created for each source file.
- One or more module files, such as `datadef.mod`, if the source file contains one or more `MODULE` statements.
- A shareable library, such as `mylib.so`, if you use the `-shared` option.
- Assembly files, such as `fred.s`, if you use the `-S` option. This creates an assembly file for each source file.

Intel Fortran Compiler: Memory Models

Applications designed to take advantage of Intel® 64 architecture can be built with one of three memory models:

- **small** (`-mcmodel=small`) (This is the **default**)
  This causes code and data to be restricted to the first 2GB of address space so that all accesses of code and data can be done with Instruction Pointer (IP)-relative addressing.

- **medium** (`-mcmodel=medium`)
  This causes code to be restricted to the first 2GB; however, there is no restriction on data. Code can be addressed with IP-relative addressing, but access of data must use absolute addressing.

- **large** (`-mcmodel=large`)
  There are no restrictions on code or data; access to both code and data uses absolute addressing.

Intel Fortran Compiler: Default integer size

Specify the default KIND for integer and logical variables:

`-integer-size 16 , -i2`
Makes default integer and logical variables 2 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=2).

`-integer-size 32 , -i4`
Makes default integer and logical variables 4 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=4). This is the **default**.

`-integer-size 64 , -i8`
Makes default integer and logical variables 8 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=8).

Intel Fortran Compiler: Default real size

Specify the default KIND for real and complex variables:

-real-size 32
Makes default real and complex variables 4 bytes long. REAL declarations are treated as single precision REAL (REAL(KIND=4)) and COMPLEX declarations are treated as COMPLEX (COMPLEX(KIND=4)).

-real-size 64, -r8
Makes default real and complex variables 8 bytes long. REAL declarations are treated as double precision REAL (REAL(KIND=8)) and COMPLEX declarations are treated as DOUBLE COMPLEX (COMPLEX(KIND=8)).

real-size 128, -r16
Makes default real and complex variables 16 bytes long. REAL declarations are treated as extended precision REAL (REAL(KIND=16)) and COMPLEX and DOUBLE COMPLEX declarations are treated as EXTENDED PRECISION COMPLEX (COMPLEX(KIND=8)).

Intel Fortran Compiler: Specifying unformatted data

Intel Fortran expects numeric data to be in native little-endian order, in which the least-significant, right-most zero bit (bit 0) or byte has a lower address than the most-significant, left-most bit (or byte).

Intel Fortran provides the capability for programs to read and write unformatted data (originally written using unformatted I/O statements) in several non-native floating-point formats and in big-endian INTEGER or floating-point format.

Intel Fortran Compiler: Specifying unformatted data

There are a number of methods for specifying a nonnative numeric format for unformatted data:

- Setting an environment variable for a specific unit number before the file is opened. The environment variable is named `FORT_CONVERT`*n*, where *n* is the unit number.

- Setting an environment variable for a specific file name extension before the file is opened. The environment variable is named `FORT_CONVERT.`*`ext`* or `FORT_CONVERT_`*`ext`*, where *`ext`* is the file name extension (suffix).

- Setting an environment variable for a set of units before the application is executed. The environment variable is named `F_UFMTENDIAN`.

Intel Fortran Compiler: Specifying unformatted data

- Specifying the CONVERT keyword in the OPEN statement for a specific unit number.
- Compiling the program with an OPTIONS statement that specifies the CONVERT=*keyword* qualifier. This method affects all unit numbers using unformatted data specified by the program.
- Compiling the program with the appropriate compiler option, which affects all unit numbers that use unformatted data specified by the program. Use the `-convert keyword` option.

If none of these methods are specified, the native LITTLE_ENDIAN format is assumed (no conversion occurs between disk and memory).

Intel Fortran Compiler: run time checks

`-check [keyword]` Checks for certain conditions at run time.

`none` Disables all check options.

`arg_temp_created` Determines whether checking occurs for actual arguments before routine calls.

`bounds` Determines whether checking occurs for array subscript and character substring expressions.

`format` Determines whether checking occurs for the data type of an item being formatted for output.

Intel Fortran Compiler: run time checks

`output_conversion` Determines whether checking occurs for the fit of data items within a designated format descriptor field.

`pointers` Determines whether checking occurs for certain disassociated or uninitialized pointers or unallocated allocatable objects.

`uninit` Determines whether checking occurs for uninitialized variables.

`all` Enables all check options.

The default is `–nocheck` No checking is performed for run-time failures.

Intel Fortran Compiler: Using C/C++ and Fortran together

- For mixed-language applications, the Intel Fortran main program can call subprograms written in C/C++ if the appropriate calling conventions are used.

- Intel Fortran subprograms can be called by C/C++ main programs. If the main program is C/C++, you need to use the `-nofor_main` compiler option to indicate this.

- You can use subprograms in static or shared libraries if the main program is written in Intel Fortran or C/C++.

Intel Fortran Compiler: Using C/C++ and Fortran together

- Fortran adds an underscore to external names; C does not.
- Fortran changes the case of external names to lowercase; C leaves them in their original case.
- Fortran passes numeric data by reference; C passes by value.
- Fortran subroutines are equivalent to C void routines.
- Fortran requires that the length of strings be passed; C is able to calculate the length based on the presence of a trailing null. Therefore, if Fortran is passing a string to a C routine, that string needs to be terminated by a null;
  for example: `"mystring“ // CHAR(0)` or `StringVar // CHAR(0)`
- For the following data types, Fortran adds a hidden first argument to contain function return values: `COMPLEX`, `REAL*16`, `CHARACTER`, and derived types.

Intel Fortran Compiler: Example Fortran calling C

```
program test2
implicit none
integer, parameter :: N = 1000000
integer :: i
real, dimension(N) :: aa, bb, cc
real add_c, add_f

do i = 1, N
   aa(i) = i
   bb(i) = 2.0 *i
end do

do i = 1, N
   cc(i) = add_f(aa(i), bb(i))
end do
print*, "cc(N/2) = ", cc(N/2)

do i = 1, N
   cc(i) = add_c(aa(i), bb(i))
end do
print*, "cc(N/2) = ", cc(N/2)
end program test2
```

Intel Fortran Compiler: Example Fortran calling C

Fortran version:

```
real function add_f(a, b)
implicit none
real :: a, b

add_f = a + b

end function add_f
```

C version:

```
float add_c_(float *a, float *b) {
   return (*a + *b);
}
```

Intel Fortran Compiler: Example Fortran calling C

Makefile:

```
test2: test2.f90 add_f.f90 add_c.c Makefile
   ifort -c test2.f90
   ifort -c add_f.f90
   icc   -c add_c.c
   ifort -o test2 test2.o add_f.o add_c.o
```

Intel Fortran Compiler: C calling Fortran

```
#include <stdio.h>
#define N 1000000
extern float add_c_(), add_f_();

void main(int argc, char *argv[]){
   float sum, aa[N], bb[N], cc[N];
   int i;

   for(i=0; i<N; i++){
      aa[i] = (double) i;
      bb[i] = (double) (2*i);
   }

   for(i=0; i<N; i++)
      cc[i] = add_c_(&aa[i], &bb[i]);
   printf("cc = %f\n",cc[N/2]);

   for(i=0; i<N; i++)
      cc[i] = add_f_(&aa[i], &bb[i]);
   printf("cc = %f\n",cc[N/2]);
}
```

Intel Fortran Compiler: C calling Fortran

Makefile:

```
test3: test3.c add_f.f90 add_c.c Makefile
   icc -c test3.c
   ifort -c add_f.f90
   icc -c add_c.c
   icc -o test3 test3.o add_f.o add_c.o
```

Link like this if the Fortran subroutines call Fortran libraries:

```
   ifort -nofor-main -o test3 test3.o add_f.o add_c.o
```

Intel Fortran Compiler: Traceback

`-traceback`

This option tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.

```
forrtl: error (73): floating divide by zero
Image              PC                Routine     Line      Source
test2              0000000000402D40  add_f_             5  add_f.f90
test2              0000000000402BD0  MAIN__            14  test2.f90
test2              0000000000402AFC  Unknown     Unknown  Unknown
libc.so.6          00002B8884EE9184  Unknown     Unknown  Unknown
test2              0000000000402A29  Unknown     Unknown  Unknown
```

Intel Fortran Compiler: Floating point exceptions

`-fpe(n)`

Allows some control over floating-point exception handling for the main program at run-time.

`0`	Floating-point invalid, divide-by-zero, and overflow exceptions are enabled. If any such exceptions occur, execution is aborted. This option sets the `-ftz` option; therefore underflow results will be set to zero unless you explicitly specify `-no-ftz`. Underflow results from SSE instructions, as well as x87 instructions, will be set to zero. By contrast, option `-ftz` only sets SSE underflow results to zero. For information about where the error occurred, use `-traceback`.
`1`	All floating-point exceptions are disabled. Underflow results from SSE instructions, as well as x87 instructions, will be set to zero.
`3`	All floating-point exceptions are disabled. Floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero, such as `-ftz`, `-O3`, or `-O2`. This setting provides full IEEE support. This is the **default**.

Intel
Math Kernel Library
MKL

Intel MKL

- The Intel Math Kernel Library (MKL) provides Fortran routines and functions that perform a wide variety of operations on vectors and matrices including sparse matrices and interval matrices.
- The MKL library also includes fast Fourier transform (FFT) functions, as well as vector mathematical and vector statistical functions with Fortran and C interfaces.
- The Linux and Windows versions of the MKL also include ScaLAPACK software and Cluster FFT software for solving respective computational problems on distributed-memory parallel computers.
- This library has been optimized for the latest generations of Intel processors.

Intel MKL

The MKL includes the following groups of routines:

- Basic Linear Algebra Subprograms (BLAS):
  - Level 1: vector operations
  - Level 2: matrix-vector operations
  - Level 3: matrix-matrix operations
- Sparse BLAS Levels 1, 2, and 3 (for operations on sparse vectors and matrices)
- LAPACK routines for solving systems of linear equations
- LAPACK routines for solving least squares problems, eigenvalue and singular value problems and Sylvester's equations
- Auxiliary and utility LAPACK routines
- Direct and Iterative Sparse Solver routines

Intel MKL

- Vector Mathematical Library (VML) functions for computing core mathematical functions on vector arguments (with Fortran and C interfaces)
- Vector Statistical Library (VSL) functions for generating vectors of pseudorandom numbers with different types of statistical distributions and for performing convolution and correlation computations
- General Fast Fourier Transform (FFT) Functions, providing fast computation of Discrete Fourier Transform via the FFT algorithms with Fortran and C interfaces
- Tools for solving partial differential equations - trigonometric transform routines and Poisson solver
- Optimization Solver routines for solving nonlinear least squares problems through the Trust-Region (TR) algorithms and computing Jacobi matrix by central differences
- GNU Multiple Precision (GMP) arithmetic functions

Intel MKL

- Basic Linear Algebra Communication Subprograms (BLACS) that are used to support a linear algebra oriented message passing interface
- Distributed memory Parallel BLAS routines (PBLAS) for Levels 1, 2, and 3
- ScaLAPACK computational, driver, and auxiliary routines (only for Linux and Windows)
- Cluster FFT functions (only for Linux and Windows)

Intel MKL: Documentation

Where is the documentation for MKL?

```
> which ifort

/sw/sdev/intel/Compiler/11.0/074/Do
  cumentation/mkl
```

- userguide.pdf 127 pages
- mklman.pdf 3460 pages!!

Intel MKL: Configuration

- Use one of the following scripts to setup the MKL environment for linux_64:

```
source /opt/intel/Compiler/11.1/038/mkl/tools/environment/mklvarsem64t.csh
source /opt/intel/Compiler/11.1/038/mkl/tools/environment/mklvarsem64t.sh
```

- Or, if modules are installed, load the appropriate module

- Do a sanity check

```
> echo $MKLROOT
/opt/intel/Compiler/11.1/038/mkl
```

Intel MKL: Libraries

Building an application using MKL requires specifying four MKL layers:

- Interface layer
- Threading layer
- Computational layer
- Real-Time Library layer

Most libraries are available in both static (`lib*.a`) and dynamic (`lib*.so`) versions.

There is also support for standard LP64 and ILP64 (for arrays larger than 2**31 -1)

Intel MKL: Complicated?

If you think this is rather complicated, you are not alone.

Intel provide a web-based linking advisor where you choose your libraries and options to specify a link line for your application:

http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor

Intel MKL: include paths

The MKL setup script or loading the MKL module will set `$MKLROOT` as well as other environment variables.

MKL include files are in `/include`

The Fortran 95 interface definitions are in `$MKLROOT/include/em64t/lp64`

The MKL libraries are in `$MKLROOT/lib/em64t`

Intel MKL: static or dynamic

Linking in dynamic libraries is generally easier than linking in static libraries.

The libraries contain circular references that must be resolved at build time when using static libraries.

Example:

```
$MKLPATH/libmkl_solver_lp64.a -Wl,--start-group $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group -openmp -lpthread
```

Intel MKL: Use the supplied examples as templates

MKL provides a comprehensive suite of examples in:
`$MKLROOT/examples`

```
> ls  $MKLROOT/examples
blas     dftc          fftw3xc  lapack95
  servicefuncs  vmlc
blas95   dftf          fftw3xf  pdepoissonc  solver
  vmlf
cblas    fftw2xc       gmp      pdepoissonf  spblas
  vslc
cdftc    fftw2x_cdft   java     pdettc       ublas
  vslf
cdftf    fftw2xf       lapack   pdettf
  versionquery
```

Intel MKL: running a MKL example

Example: build and run a program using a BLAS level 1 DAXPY routine

Copy the examples directory to your home area

```
> cd examples/blas
> ls
blas.lst  data  makefile  source
```

running `make` with no arguments or `make help` will output useful information.

```
> make soem64t function=daxpy
```

Builds and runs an example code exercising the DAXPY routine.

The output contains the complete build command!

Intel MKL: Example 1

A simple test program to call the DAXPY routine:

```
program test1
implicit none

integer, parameter :: N = 1000
real (kind=8) :: a
real (kind=8) :: x(N), y(N)

x = 12.0
y = 2.6
a = 1.7
call daxpy(N, a, x, 1, y, 1)
write(6,*) "y[8] = ", y(8)

end program test1
```

Intel MKL: Example 1

Makefile:

```
MKLINCLUDE     = $(MKLROOT)/include
MKLPATH        = $(MKLROOT)/lib/em64t
FFLAGS         = -openmp -I$(MKLINCLUDE)
LDFLAGS        = -L$(MKLPATH)        \
                 -lmkl_intel_lp64    \
                 -lmkl_intel_thread \
                 -lmkl_core
all: test1

test1: test1.o Makefile
        ifort $(FFLAGS) -o test1 test1.o $(LDFLAGS)

test1.o: test1.f90 Makefile
        ifort $(FFLAGS) -c test1.f90
```

Intel MKL: Example 2

A simple Fortran 95 test program to call the DAXPY routine:

```
program test1
use blas95
implicit none
integer, parameter :: N = 1000
real (kind=8) :: a
real (kind=8) :: x(N), y(N)

x = 12.0
y = 2.6
a = 1.7
call axpy(x, y, a)
write(6,*) "y[8] = ", y(8)

end program test1
```

Intel MKL: Example 2

Makefile:

```
MKLINCLUDE    = $(MKLROOT)/include
MKLF95INCLUDE = $(MKLINCLUDE)/em64t/lp64
MKLPATH       = $(MKLROOT)/lib/em64t

FFLAGS        = -openmp -I$(MKLINCLUDE) -I$(MKLF95INCLUDE)
LDFLAGS       = -L$(MKLPATH) -lmkl_blas95_lp64 \
                -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core
all: test1

test1: test1.o Makefile
        ifort $(FFLAGS) -o test1 test1.o $(LDFLAGS)

test1.o: test1.f90 Makefile
        ifort $(FFLAGS) -c test1.f90
```

Shared-Memory Programming with OpenMP

OpenMP: Module Objectives

After completing this module, you will be able to:

- Explain the origins of OpenMP
- Recognize OpenMP constructs and directives
- Understand what is incremental parallelism
- Write a parallel code using OpenMP

OpenMP: Motivation for OpenMP

- 1997: No standard for shared memory parallelism
  - Each SMP vendor had proprietary API
  - Portability only through MPI, PVM
- Parallel application availability
  - ISVs have big investment in existing code
  - Message passing ports are labor intensive
- OpenMP allows a partial port --- incremental parallelism

OpenMP: What Is OpenMP?

- Shared memory multiprocessing API
  - Standardizes existing practice
- Portable and standard
  - SGI, Compaq, Kuck and Assoc., DOE, IBM, Sun, HP, Intel, and so on
  - Both UNIX and NT
- First Fortran OpenMP implementation released by SGI October 1997
- First C/C++ OpenMP implementation released by SGI November 1998
- Intel version 11 Compilers support OpenMP 3.0

OpenMP: Parallel Programming Execution Model

- OpenMP uses a fork and join execution model
- Execution begins in a single master thread
- Parallel regions are executed by a team of threads
- Execution returns to a single thread at the end of a parallel region
- OpenMP provides constructs for
  - Data parallelism---Loop level
  - Functional parallelism

OpenMP: OpenMP Overview

- Scalable: fine and coarse grain parallelism
- Emphasis on performance
- Exploit strengths of shared-memory
- Directive-based (plus library calls and environment variables)

OpenMP: Directive Sentinels

Fortran

```
!$OMP
C$OMP
*$OMP
```

- **C/C++**

```
#pragma omp directive
{ structured block }
```

OpenMP: Conditional Compilation

OpenMP allows certain statements to be conditionally compiled

Fortran

`!$` Fixed or free form

`C$` Fixed form

When compiled with the OpenMP option these sentinels are replaced in the code with two spaces.

```
!$     record = ( loopcnt - 1 ) *
!$    + myid
```

C/C++

Use the macro `_OPENMP`

Can NOT be used with `#define` or `#undef`

OpenMP: Parallel Regions

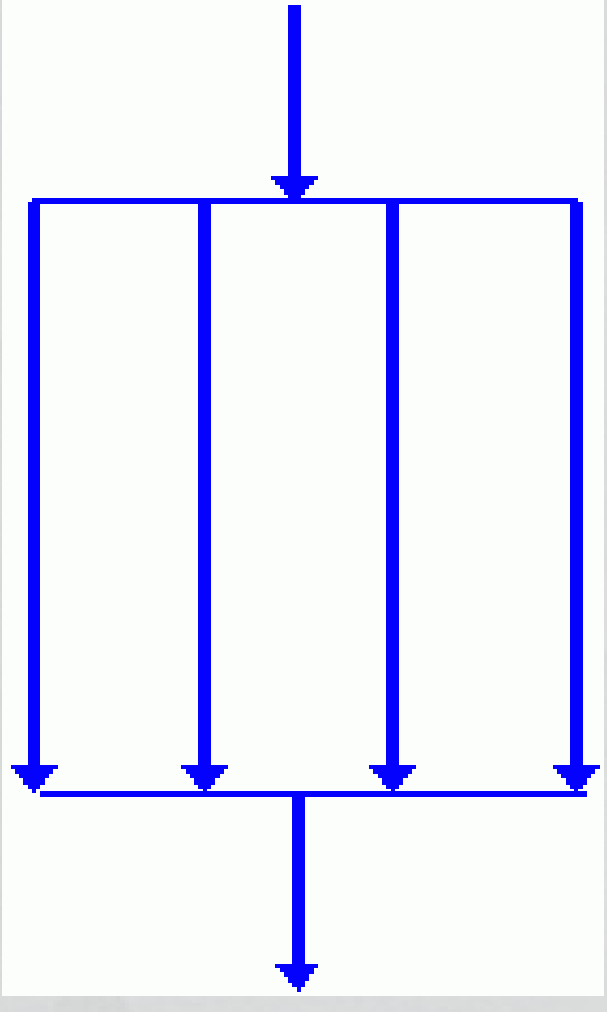
Fortran

```
!$omp parallel
...
!$omp end parallel
```

C/C++

```
#pragma omp
  parallel
{ structured block
  }
```

Be careful of C++ *throws*



OpenMP: Parallel Regions, Optional Clauses

The following optional clauses may be added to the omp parallel directive:

- `private ( list )`
- `shared ( list )`
- `default ( private | shared | none )`
- `firstprivate ( list )`
- `reduction ( ( operator | intrinsic ) : list )`
  - `operator` must not be overloaded (C++)

- `if ( scalar_logical_expression )`
- `copyin ( list )`

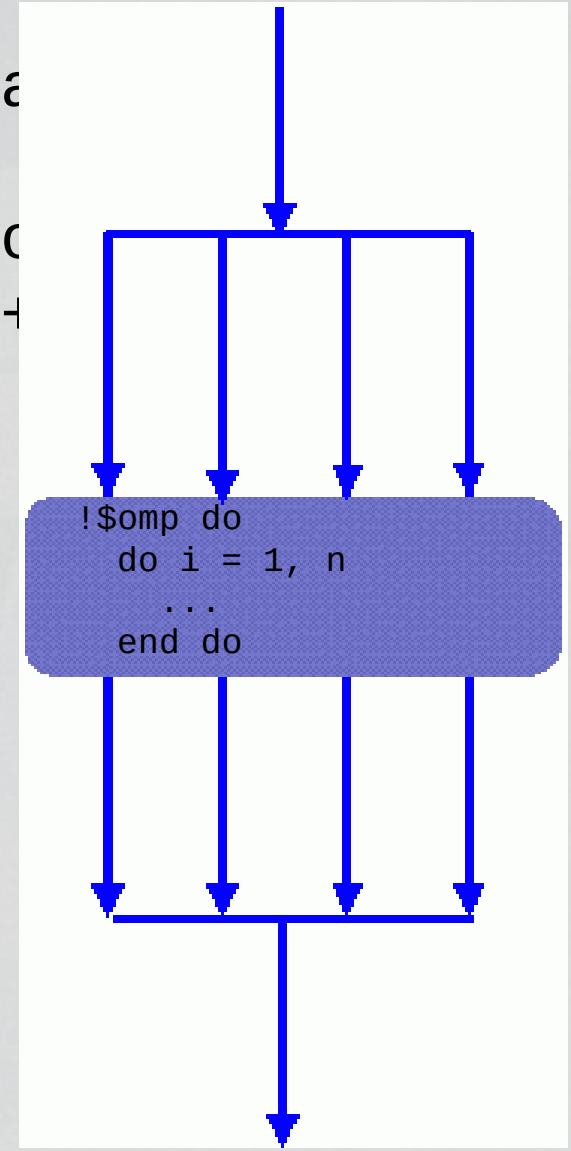
OpenMP: Work Sharing: DO / FOR

Fortran

```
!$omp parallel
...
!$omp do
do i=1,n
  a(i)=0
enddo
!$omp end do
...
!$omp end parallel
```

C/C++

```
#pragma omp para
{ ...
  #pragma omp fo
  for(i=0;i<n;i+
     a[i]=0.0
}
```



OpenMP: Work Sharing: DO / FOR continued

The following clauses can be added to the DO/FOR construct:

```
lastprivate ( list )
schedule ( type [ , chunk ] )
```

where `type` is :

- `static`
- `dynamic`
- `guided`
- `runtime`
- `auto`

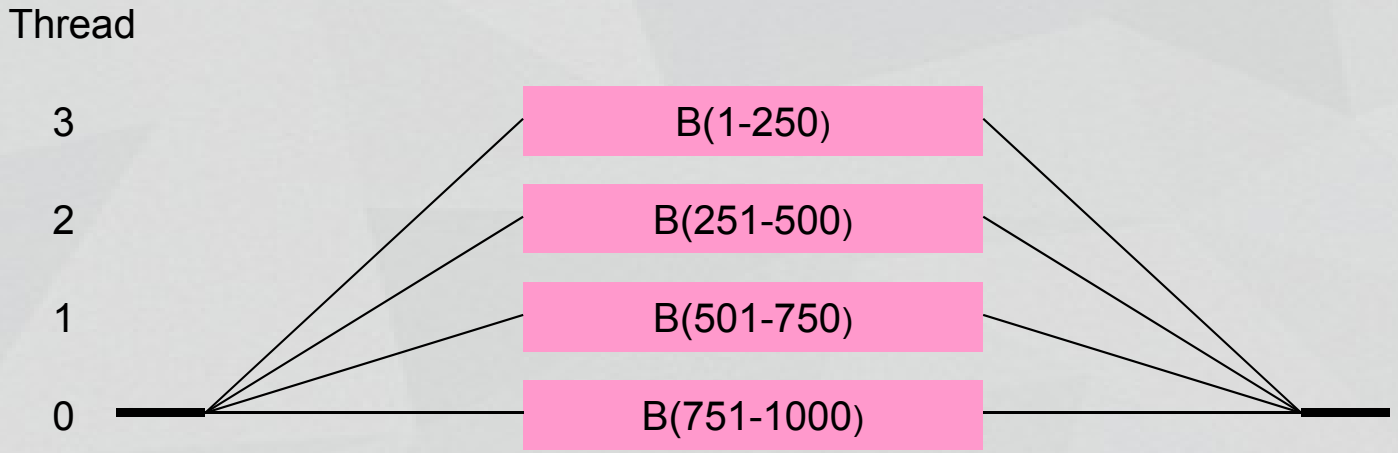
```
ordered
```

There is an optional `end do` directive, with has an optional clause:

- `nowait`

OpenMP: Loop scheduling types

If `schedule` is `static` and `chunksize` is not specified, each thread gets one chunk.

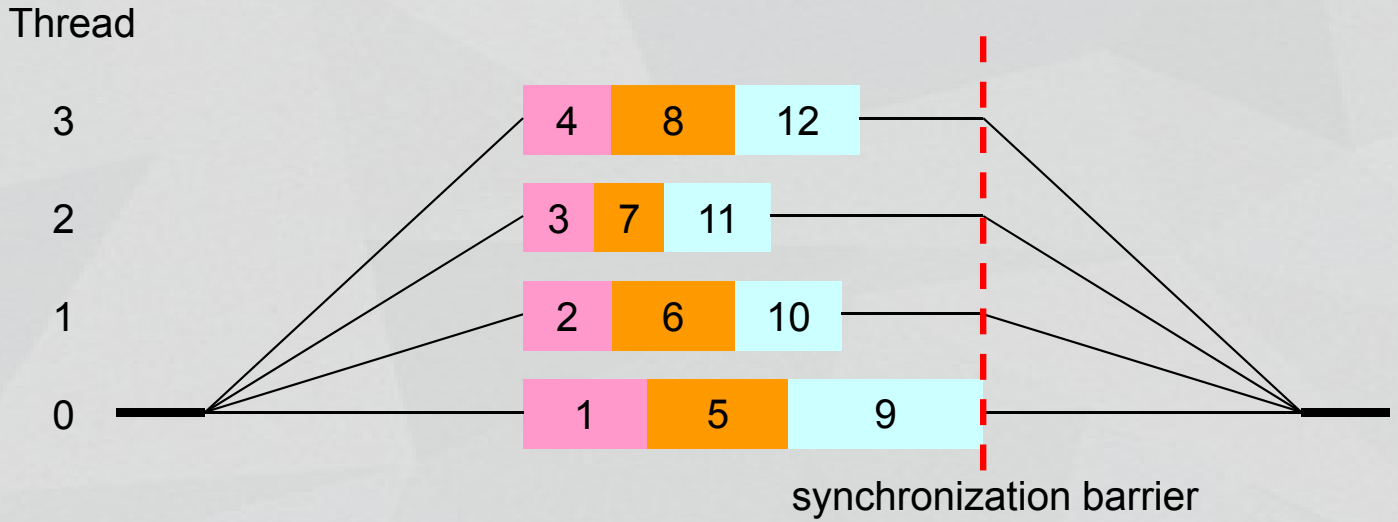



```
!$omp parallel
...
!$omp do schedule(static)
   do i = 1, 1000
   ! code block B
   enddo
...
!$omp end parallel
```

```
#pragma omp parallel
{ ...
        #pragma omp for schedule(static)
        for (i=0;i<1000;i++)
        /*code block B*/
}
```

OpenMP: Loop scheduling types (continued)

- `static` with `chunksize` distributes work cyclically in blocks of `chunksize` iterations

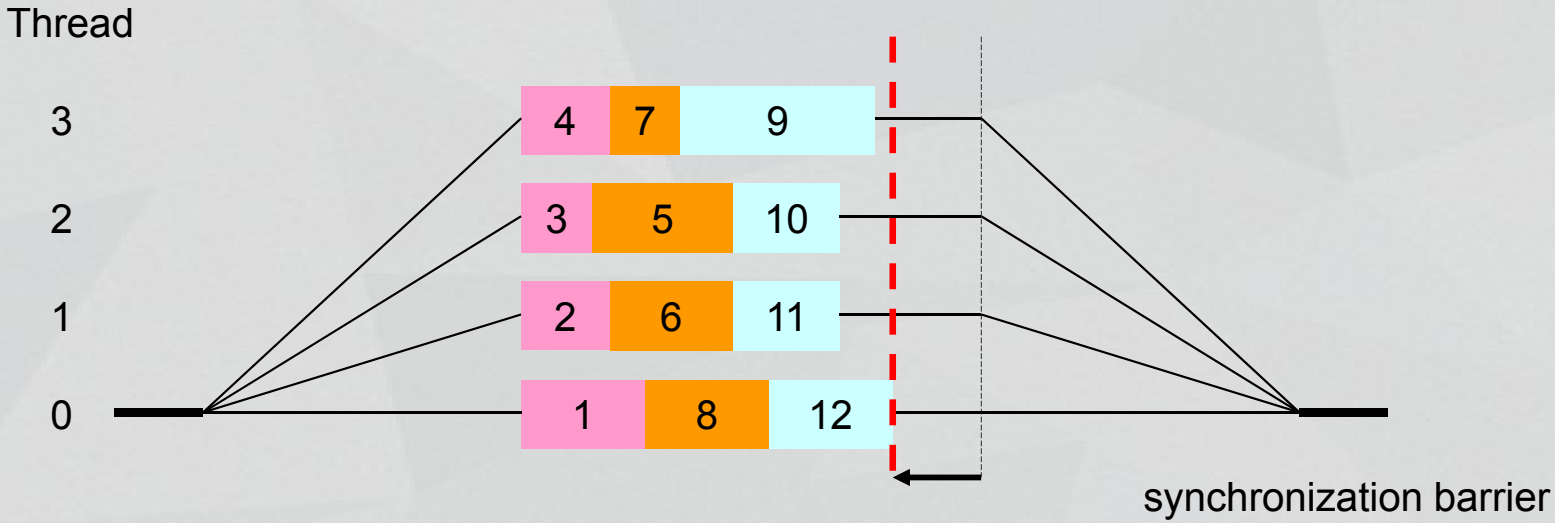



```
!$omp parallel
...
!$omp do schedule(static, 100)
   do i = 1, 1200
   ! code block B
   enddo
...
!$omp end parallel
```

```
#pragma omp parallel
{ ...
   #pragma omp schedule(static, 100)
   for (i=0;i<1200;i++)
   /*code block B*/
}
```

OpenMP: Loop scheduling types (continued)

- `dynamic` dynamically allocates work in blocks of `chunksize` iterations




```
!$omp parallel
...
!$omp do schedule(dynamic, 100)
   do i = 1, 1200
   ! code block B
   enddo
...
!$omp end parallel
```

```
#pragma omp parallel
{ ...
   #pragma omp schedule(dynamic, 100)
   for (i=0;i<1200;i++)
   /*code block B*/
}
```

OpenMP: Loop scheduling types (continued)

- `guided` is dynamic scheduling that starts with large chunks and ends with smaller chunks; `chunksize` is the minimum number of iterations assigned.

```
!$omp parallel
...
!$omp do schedule(guided, 100)
   do i = 1, 1200
   ! code block B
   enddo
...
!$omp end parallel
```

```
#pragma omp parallel
{ ...
      #pragma omp for schedule(guided, 100)
      for (i=0;i<1200;i++)
        /*code block B*/
}
```

OpenMP: Loop scheduling types (continued)

- `runtime` indicates that the scheduling type and chunk size are to be set at run time by the environment variable `OMP_SCHEDULE`.

- `auto` delegates the decision regarding scheduling to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.

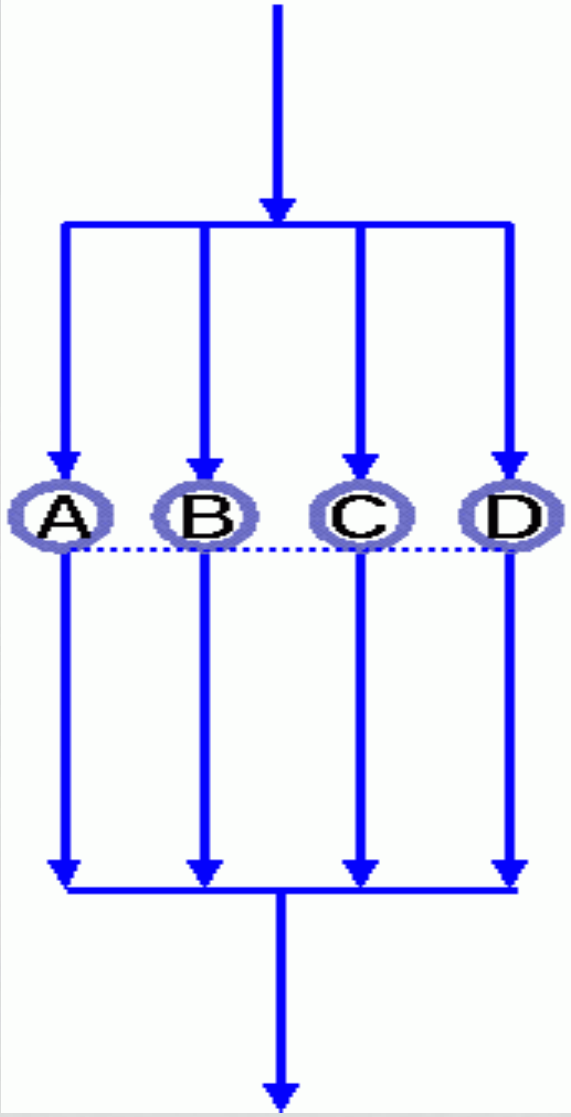
OpenMP: Work Sharing: Sections

```
!$omp parallel
      ...
!$omp sections

!$omp section
      A...
!$omp section
      B...
!$omp section
      C...
!$omp section
      D...
!$omp end sections

      ...
!$omp end parallel
```

```
#pragma omp parallel
{ ...
#pragma omp sections
  {
#pragma omp section
    {structured block A}
#pragma omp section
    {structured block B}
#pragma omp section
    {structured block C}
#pragma omp section
    {structured block D}
  }
  ...
}
```

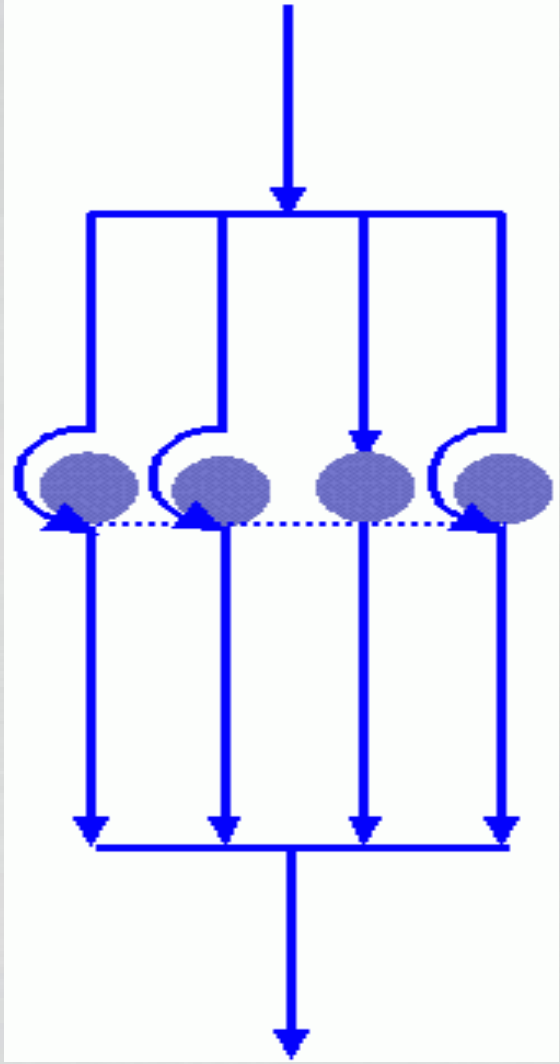


OpenMP: Work Sharing: Single

```
$omp parallel
      ...
!$omp single
      ...
!$omp end single [nowait]
      ...
!$omp end parallel
```

```
#pragma omp parallel
{ ...
   #pragma single
   {structured block}
  ...
                }
```

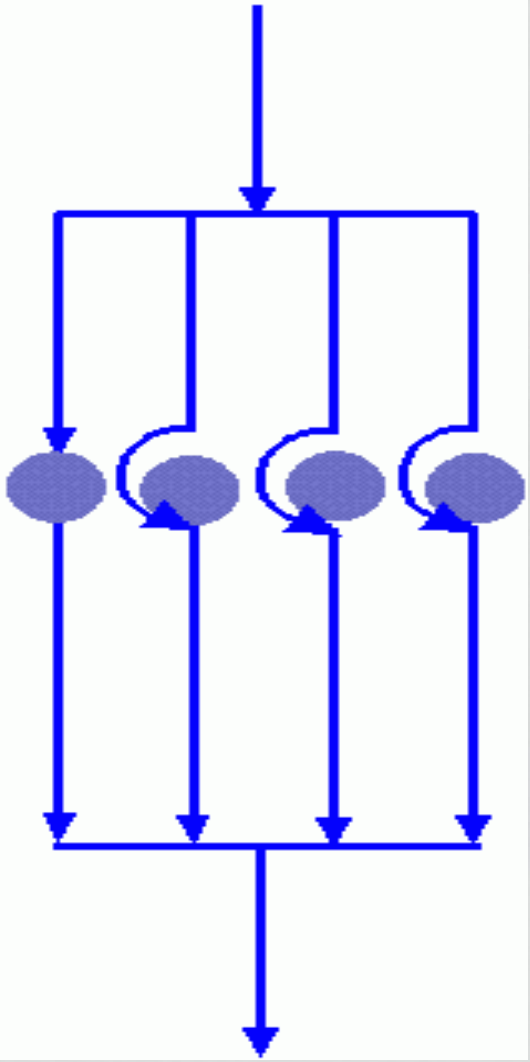
There is an implicit barrier at the end of the `single` section.



OpenMP: Work Sharing: Master

```
$omp parallel
      ...
!$omp master
      ...
!$omp end master
      ...
!$omp end parallel
```

```
#pragma omp parallel
{ ...
   #pragma master
   {structured block}
  ...
                }
```



Unlike the `single` construct, there is NO implicit barrier at the end of the `master` section.

OpenMP: Workshare Construct (Fortran only)

The workshare construct divides the execution of the enclosed structured block into separate units of work, and causes the threads of the team to share the work such that each unit is executed only once by one thread, in the context of its implicit task.

```
!$omp workshare
   structured-block
!$omp end workshare [nowait]
```

OpenMP: Workshare Construct (Fortran only)

The enclosed structured block must consist of only the following:

- array assignments
- scalar assignments
- FORALL statements
- FORALL constructs
- WHERE statements
- WHERE constructs
- atomic constructs
- critical constructs
- parallel constructs

OpenMP: Workshare Construct (Fortran only)

Example:

```
!$omp parallel
!$omp workshare
  A = 0.0
  B = 0.0
  C = sin(A) + cos(B)
  where (val > 0.0
   A = log(B)
  elsewhere
   A = 0.0
  end where
!$omp end workshare
!$omp end parallel
```

OpenMP: Tasks

OpenMP tasks were introduced to allow dynamic worksharing not possible using the OpenMP sections construct.

This allows irregular problems to parallelized.

C:

```
#pragma omp task [clauses]
  structured block
```

Fortran:

```
!$omp task [clauses]
  structured block
!$omp end task
```

OpenMP: taskwait

The `taskwait` construct specifies a wait on the completion of child tasks generated since the beginning of the current task.

C:

```
#pragma omp taskwait newline
```

Fortran:

```
!$omp taskwait
```

OpenMP: task example

C:

```
#pragma omp parallel
{
    #pragma omp single
   {
    while (work_to_do)
    #pragma omp task
    {
       do_work();
    }
    #pragma taskwait
   }
}
```

OpenMP: Work Sharing: Shorthand

```
!$omp parallel do
      do i=1,n
      ...
      enddo
!$omp end parallel do
```

```
!$omp parallel sections
!$omp section
      ...
!$omp section
      ...
!$omp end parallel sections
```

```
#pragma omp parallel for
   for (...)
                {
```

```
#pragma omp parallel sections
{
#pragma omp section
   { structured block }
   #pragma omp section
   { structured block }
```

OpenMP: Work Sharing: Shorthand (continued)

It is common to see the following in OpenMP programs

```
!$omp parallel do
    (do loop 1)
!$omp end parallel do
...
!$omp parallel do
    (do loop 2)
!$omp end parallel do
...
!$omp parallel do
    (do loop 3)
!$omp end parallel do
```

This results in possibly swapping from single to multi thread mode at each parallel region.

OpenMP: Work Sharing: Shorthand (continued)

Where possible, arrange the code so all parallel work can be done in one region, and the sequential code before or after the parallel region:

```
...
!$omp parallel
!$omp do
    (do loop 1)
!$omp do
    (do loop 2)
!$omp do
    (do loop 3)
!$omp end parallel do
...
```

If it is necessary to have sequential code between the loops, consider using the `single` clause around this code.

OpenMP: Orphaning

The worksharing constructs may be outside lexical scope of parallel region:

```
!$omp parallel
      ...
      call foo (...)
      ...
!$omp end parallel

subroutine foo (...)
...
!$omp do
do i=1,n
  ...
enddo
end subroutine foo
```

OpenMP: Data Scoping Clauses

```
common /mycommon/ f1, f2
!$omp threadprivate (/mycommon/)
real a(n), b(n), sum
!$omp parallel shared(a) private(b)
      ...
!$omp end parallel

!$omp parallel default(private) shared(b) reduction(+:sum)
      ...
!$omp  end parallel
```

```
firstprivate, lastprivate
reduction
default (shared|private)
threadprivate
```

OpenMP: Synchronization

- Barriers

```
!$omp barrier                    #pragma omp barrier
```

- Critical sections

```
!$omp critical [printlock]    #pragma omp critical
   [name]
!$omp end critical                  { structured block
   }
```

- Lock library routines

```
omp_set_lock(var)
omp_unset_lock(var)
...
```

OpenMP: Synchronization (continued)

- Atomic

```
!$omp atomic                    #pragma omp atomic
count = count + 1               count += 1;
```

- Flush

```
!$omp flush [(list}]            #pragma omp flush [(list)]
```

  - Maintain memory consistency
  - Restore/reload thread-visible variables to/from memory
  - Useful for custom synchronization between threads

OpenMP: Synchronization: DO/FOR ordered

```
!$omp do ordered
      do i = ...
      ...
!$omp ordered
      print *,i,result(i)
!$omp end ordered
      ...
      enddo
!$omp end do
```

```
#pragma omp for ordered
  for (...)
  { ...
    #pragma omp ordered
    { structured block }
    ...
  }
```

OpenMP: Intel OpenMP compiler options

Version 11.0.076 of the Intel compiler supports OpenMP API Version 3.0 (http://www.openmp.org click Specifications).

`-openmp`	This option enables the parallelizer to generate multi-threaded code based on the OpenMP* directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems.
`-openmp-report`	This option controls the OpenMP parallelizer's level of diagnostic messages. To use this option, you must also specify `-openmp`.
`-openmp-stubs`	This option enables compilation of OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked.
`-openmp-profile`	This option enables analysis of OpenMP* applications. To use this option, you must have previously installed Intel® Thread Profiler, which is one of the Intel® Threading Analysis Tools.
`-openmp-task`	The option lets you choose an OpenMP tasking model. To use this option, you must also specify option `–openmp`.

OpenMP: mixed options

When both `-openmp` and `-parallel` are both specified on the command line, the `parallel` option is only applied in loop nests that do not have OpenMP directives.

For loop nests with OpenMP directives, only the `-openmp` option is applied.

OpenMP: OpenMP environment variables

`OMP_NUM_THREADS`	Sets the maximum number of threads to use for OpenMP parallel regions if no other value is specified in the application. This environment variable applies to both `-openmp` and `-parallel`
`OMP_SCHEDULE`	Sets the run-time schedule type and an optional chunk size. Default is `static` with no chunksize.
`OMP_DYNAMIC`	Enables (1) or disables (0) the dynamic adjustment of the number of threads. Default is disabled.
`OMP_NESTED`	Enables (1) or disables (0)nested parallelism. Default is disabled.

OpenMP: OpenMP environment variables

`OMP_STACKSIZE`	Sets the memory to allocate for each OpenMP thread to use for a private stack. Use the optional suffixes: B (bytes), K (Kilobytes), M (Megabytes), G (Gigabytes), or T (Terabytes) to specify the units. If only value is specified then size is assumed to be K (Kilobytes). Default is 4M.
`OMP_MAX_ACTIVE_LEVELS`	Limits the number of simultaneously executing threads in an OpenMP program.
`OMP_THREAD_LIMIT`	Limits the number of simultaneously executing threads in an OpenMP* program.

OpenMP: KMP environment variables

`KMP_ALL_THREADS`	Limits the number of simultaneously executing threads in an OpenMP* program.
`KMP_BLOCKTIME`	Sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping. Use the optional character suffixes: s (seconds), m (minutes), h (hours), or d (days) to specify the units. Specify `infinite` for an unlimited wait time. Default is 200 ms.
`KMP_LIBRARY`	Selects the OpenMP run-time library execution mode. The options for the variable value are `throughput`, `turnaround`, and `serial`. Default is `throughput`.

OpenMP: KMP environment variables

KMP_STACKSIZE	Sets the number of bytes to allocate for each OpenMP* thread to use as the private stack for the thread. Default is 4m
KMP_MONITOR_STACKSIZE	Sets the number of bytes to allocate for the monitor thread, which is used for book-keeping during program execution. Default is max (32k, system minimum thread stack size)
KMP_VERSION	Enables (1) or disables (0) the printing of OpenMP run-time library version information during program execution. Default is disabled.

OpenMP: KMP environment variables

KMP_AFFINITY	Enables run-time library to bind threads to physical processing units. See Thread Affinity Interface for more information on the default and the affect this environment variable has on the parallel environment.
KMP_SETTINGS	Enables (1) or disables (0) the printing OpenMP run-time library environment variables during program execution. Default is disabled.
KMP_CPUINFO_FILE	Specifies an alternate file name for file containing machine topology description. The file must be in the same format as /proc/cpuinfo.

OpenMP: Run-Time Library Routines

- Execution Environment Routines
- Lock Routines
- Timing Routines
- Intel Extension Routines

OpenMP: Execution Environment Library Routines

- `void` **`omp_set_num_threads`**`(int` *`nthreads`*`)`
- `int` **`omp_get_num_threads`**`(void)`
- `int` **`omp_get_max_threads`**`(void)`
- `int` **`omp_get_thread_num`**`(void)`
- `int` **`omp_get_num_procs`**`(void)`
- `int` **`omp_in_parallel`**`(void)`
- `void` **`omp_set_schedule`**`(omp_sched_t kind,int modifier)`
- `void` **`omp_get_schedule`**`(omp_sched_t *kind,int *modifier)`
- `int` **`omp_get_thread_limit`**`(void)`
- ….plus calls for nested thread levels

OpenMP: Lock Routines

- `void` **`omp_init_lock`**`(omp_lock_t` *`lock`*`)`
- `void` **`omp_destroy_lock`**`(omp_lock_t` *`lock`*`)`
- `void` **`omp_set_lock`**`(omp_lock_t` *`lock`*`)`
- `int` **`omp_test_lock`**`(omp_lock_t` *`lock`*`)`
- …. plus calls for nested locks

OpenMP: Timing Routines

- `double` **`omp_get_wtime`**`(void)`

  Returns a double precision value equal to the elapsed wall clock time (in seconds) relative to an arbitrary reference time. The reference time does not change during program execution.

- `double` **`omp_get_wtick`**`(void)`

  Returns a double precision value equal to the number of seconds between successive clock ticks.

OpenMP: Intel Extension Routines: Execution Environment

- `void **kmp_set_defaults**(char const *)`
- `void **kmp_set_library_throughput**()`
- `void **kmp_set_library_turnaround**()`
- `void **kmp_set_library_serial**()`
- `void **kmp_set_library**(int)`
- `int **kmp_get_library**()`

OpenMP: Intel Extension Routines: Stack Size

- `size_t` **`kmp_get_stacksize_s`**`()`

  Returns the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can be changed with `kmp_set_stacksize_s()` routine, prior to the first parallel region or via the KMP_STACKSIZE environment variable.

- `void` **`kmp_set_stacksize_s`**`(size_t` *`size`*`)`

  Sets to `size` the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can also be set via the KMP_STACKSIZE environment variable. In order for `kmp_set_stacksize_s()` to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region in the program.

Deprecated:

- `int` **`kmp_get_stacksize`**`()`
- `void` **`kmp_set_stacksize`**`(int` *`size`*`)`

OpenMP: Intel Extension Routines: Memory Allocation

- `void* kmp_malloc(size_t size)`
  Allocate memory block of `size` bytes from thread-local heap.

- `void* kmp_calloc(size_t nelem, size_t elsize)`
  Allocate array of `nelem` elements of size *elsize* from thread-local heap.

- `void* kmp_realloc(void* ptr, size_t size)`
  Reallocate memory block at address `ptr` and `size` bytes from thread-local heap.

- `void* kmp_free(void* ptr)`
  Free memory block at address `ptr` from thread-local heap.
  Memory must have been previously allocated with `kmp_malloc()`, `kmp_calloc()`, or `kmp_realloc()`.

OpenMP: Intel Extension Routines: Thread Sleep

- `int` **`kmp_get_blocktime`**`(void)`

  Returns the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping, as set either by the `KMP_BLOCKTIME` environment variable or by `kmp_set_blocktime()`.

- `void` **`kmp_set_blocktime`**`(int` *`msec`*`)`

  Sets the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping. This routine affects the block time setting for the calling thread and any OpenMP team threads formed by the calling thread. The routine does not affect the block time for any other threads.

OpenMP: Useful Information

On Clovertown, Harpertown, OpenMP performance is very dependent on thread placement and the number of threads per core.

The environment variable: `KMP_AFFINITY verbose,none` outputs very usefull information regarding thread placement.

Use `top` and then press `1` to display information for each core.

The options for `KMP_AFFINITY` such as `compact` and `scatter` will place threads.

Also look at using `taskset -c a.out`. (`man taskset`) to place threads explicitly.

Compiling for Shared-Memory Parallelism

Shared-Memory Parallelism

Module Objectives:

Introduce features for compiling for shared-memory parallelism

Introduce methods of breaking data dependencies

Introduce parallelization performance issues

Use the compilers to parallelize serial code

Compiling for Shared-Memory Parallelism

- The Intel C/C++, and Fortran compilers can generate parallel code:
  - Enabled by using `-openmp` or `-parallel` options when compiling
  - Interprets directives for parallelization
  - Introduces parallelism into code without jeopardizing serial execution

- OpenMP Directives are ignored when `-openmp` is not specified.

- Generation of parallel code for loops is attempted when `-parallel` is used.

- When both `-openmp` and `-parallel` are used no automatic parallelization is attempted in routines that already have OpenMP directives.

- The `-openmp` option in Fortran sets the `-auto` option to ensure stack allocation of all local variables.

Identifying Parallel Opportunities in Existing Code

- Loops with potential for parallelism
  - Loops without data dependencies
  - Loops with data dependencies because of
    - Temporary variables
    - Reductions
    - Nested loops
    - Function calls or subroutines
- Loops without potential for parallelism
  - Premature exit
  - Too few iterations
  - Programming effort to avoid data dependencies is too great

Parallelizing Loops Without Data Dependencies

- Simple C loop:

```
for (i = 0; i < max; i++) {
 a[i] = b[i] + c[i];
}
```

- Simple Fortran loop:

```
do i = 1, max
 a(i) = b(i) + c(i)
enddo
```

- Variable types that are not specified default to `shared`.

Parallelizing Loops Without Data Dependencies (continued)

- Simple C loop explicitly parallelised using OpenMP:

```
#pragma omp parallel for shared(a, b, c, max)
  private(i)
   for (i = 0; i < max; i++) {
      a[i] = b[i] + c[i];
   }
```

- Simple Fortran loop explicitly parallelised using OpenMP:

```
c$omp parallel do shared(a, b, c, max),
  lastprivate(i)
      do i = 1, max
         a(i) = b(i) + c(i)
      enddo
```

- Loop index must be `private` (default) or `lastprivate`

Parallelizing Loops With Temporary Variables

- Temporary variables can create data dependencies if they are shared variables:

```
for (i = 0; i < n; i++) {
   tmp = a[i];
   a[i] = b[i];
   b[i] = tmp;
}

#pragma omp parallel for shared(a, b, n) private(i, tmp)
    for (i = 0; i < n; i++) {
       tmp = a[i];
       a[i] = b[i];
       b[i] = tmp;
    }
```

Parallelizing Loops With Temporary Variables

- Use `lastprivate` if variable can be treated as `private`, but is to be used after the loop. By default, index variables are `private`.

```
c$omp parallel do lastprivate(i, x)
  shared(a, b, n)
        do i = 1, n
           x = a(i) + b(i)
        enddo
        print(*) i, x
```

Parallelizing Reductions

- Generating a single value from the elements of an array is referred to as a reduction
  - There is a data dependence because the same memory location is written to over multiple loop iterations

- Use the `reduction` clause in the corresponding directive or pragma

Parallelizing Nested Loops

- Nested loops provide several options as to which loop to parallelize.
- Reorder the loops to:
  - Put the most amount of work in each iteration
  - Remove all data dependencies

Example: Data dependence on `a[i][j]` (parallelizing the `k` loop):

```
for (k=0; k<n; k++)
   for (j=0; j<n; j++)
      for (i=0; i<n; i++)
         a[i][j] = a[i][j] + b[i][k];
```

Reordering the loops removes the data dependence (parallelizing the `i` loop):

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      for (k=0; k<n; k++)
         a[i][j] = a[i][j] + b[i][k];
```

Parallelizing Loops With Subroutines and Functions

- Make sure called routines have no side effects:
  - Modifies or uses only shared variables indexed by loop control variable or in a critical region.
  - Must not use static variables.
  - Each call from a thread must be independent of any call from another thread.
- Simple test for side effects
  - If the function could be transformed into inlined code, it is probably safe.
- Parallel-safe functions:
  - All Fortran intrinsic functions
  - Functions in the C standard library
  - All functions in the math library

Unparallelizable Loops

- Recurrence: loop iteration order not changeable

```
DO I = 2,N
   X(I) = X(I-1) + Y(I)
ENDDO
```

- Exit branch

```
DO I = 2,N
   ! some work
   IF ( VALUE .GT. MAX ) GOTO 999
ENDDO
```

Unparallelizable Loops (continued)

- Loops with too few iterations (not worth parallelizing)

```
DO I = 1,4
    X(I) = 1
ENDDO
```

- Loops with calls to functions with side effects

```
DO I = 2,N
   ! some work
   call unsafe_function
ENDDO
```

Reducing Parallelization Overhead

- Measuring parallelization overhead
  - Time serial run
  - Calculate one thread of parallel program
    - Time parallel program running single threaded
  - Compare single-thread time to original serial run
- Only parallelize a region if the performance gain is more than the overhead
- Use a conditional parallelization modifier to control when a region is parallelized

Conditional Parallelization

- Can conditionally parallelize a region based on how much work it does
- Most often used with loops

- C/C++

```
#pragma omp parallel for private(i) if (max > 50)
    for (i = 0; i < max; i++) {
        /* loop body */
    }
```

- Fortran

```
C$omp parallel do private(i) shared(n) if (n .GT. 50)
      do i = 1, n
            ! work
      enddo
```

Load Balancing

- Keep all threads busy
- Find load imbalances
- Balance work done in independent blocks
- Examine loops to determine the best scheduling type for loops

Process Spin Time

- Processes spin when waiting for more work
  - Ready to get new work immediately
  - Wastes CPU time if no new work is available
- Set the amount of time to spin before blocking
- Reduce spin time if parallel regions are isolated or widely spaced
- Increase spin time for parallel regions that are clustered in one portion of the program

Guidelines for Incremental Parallelization

- Profile serial code
  - Determine regions in which time is significant
- Look for opportunities in significant regions
  - Introduce parallel loops
  - Identify potential independent blocks
  - Rearrange code for best scheduling
- Modify dependent code blocks
  - Identify suitable code blocks
  - Isolate dependence in critical section or single-processor blocks
- Guard against dependence
  - Synchronize

Automatic Parallelization Limitations

- C/C++
  - Only analyzes certain for loops
    - for loops using explicit array notation: `array[index]`
    - for loops using pointer increment notation: `*var++`
  - Cannot analyze for loops using pointer arithmetic notation: `*(var + index)`
  - Cannot analyze while or do/while loops
  - Does not look for blocks of code to be run in parallel

- Fortran
  - Only analyzes DO loops

Fortran Example

```
program test1
      real*4 a(0:9999), b(0:9999), total

      total = 0.0
      do i = 0, 9999
         b(i) = i
      enddo

      do i=0, 9999
         a(i) = b(i) + 1.0
      enddo

      do i=0, 9999
         a(i) = safety_unknown(a,i)
      enddo

      do i=0, 9999
         total = total + a(i)
      enddo

      write(*,*) total
end program test1
```

Fortran Example (continued)

```
% ifort -parallel –par-report3 -c test1.f

test1.f90(5): (col. 7) remark: DISTRIBUTED LOOP WAS AUTO-PARALLELIZED.
test1.f90(5): (col. 7) remark: loop was not parallelized: insufficient
   computational work.
test1.f90(9): (col. 7) remark: LOOP WAS AUTO-PARALLELIZED.
test1.f90(17): (col. 7) remark: DISTRIBUTED LOOP WAS AUTO-PARALLELIZED.
test1.f90(17): (col. 7) remark: loop was not parallelized: insufficient
   computational work.
test1.f90(5): (col. 7) remark: PARTIAL LOOP WAS VECTORIZED.
test1.f90(9): (col. 7) remark: LOOP WAS VECTORIZED.
test1.f90(17): (col. 7) remark: PARTIAL LOOP WAS VECTORIZED.
test1.f90(5): (col. 7) remark: PARTIAL LOOP WAS VECTORIZED.
test1.f90(9): (col. 7) remark: LOOP WAS VECTORIZED.
test1.f90(17): (col. 7) remark: PARTIAL LOOP WAS VECTORIZED.
```

C Example

```
main() {
        float a[10000], b[10000], total=0.0;
        int i;

        for (i=0; i <= 9999; i++)
                b[i] = i;

        for (i=0; i <= 9999; i++)
                a[i] = b[i] + 1.0;

        for (i=0; i <= 9999; i++)
                a[i] = safety_unknown(a,i);

        for (i=0; i <= 9999; i++)
                total = total + a[i-1];

        printf("%d\n", total);
}
```

C Example (continued)

```
> icc -parallel -par-report3 -c test2.c
   procedure: main
   procedure: main
test2.c(5): (col. 9) remark: LOOP WAS AUTO-PARALLELIZED.
test2.c(8): (col. 9) remark: LOOP WAS AUTO-PARALLELIZED.
test2.c(11): (col. 9) remark: loop was not parallelized: existence of
   parallel dependence.
test2.c(14): (col. 9) remark: LOOP WAS AUTO-PARALLELIZED.
test2.c(5): (col. 9) remark: LOOP WAS VECTORIZED.
test2.c(8): (col. 9) remark: LOOP WAS VECTORIZED.
test2.c(14): (col. 9) remark: LOOP WAS VECTORIZED.
test2.c(5): (col. 9) remark: LOOP WAS VECTORIZED.
test2.c(8): (col. 9) remark: LOOP WAS VECTORIZED.
test2.c(14): (col. 9) remark: LOOP WAS VECTORIZED.
```

Strategy for Using -parallel

- Profile code to determine areas of most performance gain
- Look at output of `-par_report` for more parallelization opportunities
  - The auto-parallelizer may need more information
- Insert directives or enable compiler options to help the auto-parallelizer
  - Information to parallelize/optimize more
  - Indicate which loops are unimportant
- Check that the program still generates the same results
- Repeat the above steps as many times as needed

Controlling the Parallelization Analysis

- Directives to control parallelization:
  - C/C++
    ```
    #pragma parallel (always)
    #pragma noparallel
    ```
  - Fortran
    ```
    !DEC$ PARALLEL (ALWAYS)
    !DEC$ NOPARALLEL
    ```

Debugging with -parallel

- Debug first running single-threaded

```
> setenv OMP_NUM_THREADS 1
> setenv KMP_LIBRARY serial
```

- Using debuggers on transformed code:
  - Compile with –g or -debug

```
> ifort -g –O2 -parallel file.f
```

- Avoid transformations that confuse the debugger (inlining and loop unrolling)

- Parallel regions, loops and sections get transformed into functions

```
__<subprogram_name>_<line_no>__par_region<seq_no>
__<subprogram_name>_<line_no>__par_loop<seq_no>
__<subprogram_name>_<line_no>__par_section<seq_no>
```

OpenMP:

Useful OpenMP links:

OpenMP reference:

www.openmp.org

Useful OpenMP tutorial and reference:

https://computing.llnl.gov/tutorials/openMP/

Distributed Memory Programming MPI

Distributed Memory Programming: Module Objectives

- Describe the Message Passing Interface (MPI) programming model as implemented by Intel MPI for Intel X86_64.

- Build and execute MPI programs on the Intel X86_64 using the Intel MPI library.

MPI: Standard

- MPI is a portable message passing style that was first standardized in 1994
  - Built on experience with the Parallel Virtual Machine model
  - Available on all HPC vendor platforms today
  - Most widely used HPC parallel programming style
- Intel MPI library is a full implementation of the MPI-2 standard

MPI: What is MPI?

- Widely accepted standard for developing parallel programs with messages.

- The communication model used on MPPs with distributed memory:
  - Can also be used in shared-memory machines
  - Provides source code portability
  - Allows efficient implementation
  - Contains a rich set of routines, yet most programs can run using only a few of the routines

MPI: What is MPI? (continued)

- Target implementations:
  - MIMD models
  - Distributed-memory clusters and multiprocessors
  - Shared-memory platforms
  - Support for virtual process topologies
  - No dynamic task spawning (uses a fixed number of available processes)

MPI: What is MPI? (continued)

- Explicit communication:
  - The user inserts communication (MPI) calls into the program explicitly
  - Offers point-to-point (PE-to-PE, workstation-to-workstation) or global (one-to-all, all-to-one, all-to-all)

MPI: What is a Message?

- A message consists of:
  - Data to be passed (optional, but almost always present)
  - A user-defined integer “tag”
- A message exists within a communicator
- Contents and meaning of a message are determined by the application:
  - Data type to be passed
  - How much data to be passed
  - How many times a message is to be passed

MPI: Some Messaging Terms

- Nonblocking – the function may return before the operation completes and before the user is allowed to reuse resources (such as buffers) specified in the call
- Blocking – a return from the function indicates that the user is allowed to reuse resources specified in the call
- Local – completion of the function depends only on the local executing process
- Nonlocal – completion of the operation may require the execution of some MPI function on another process

MPI: Processes

- An MPI program consists of autonomous processes
  - Each executes its own code (MIMD style)
  - The code need not be identical
- Processes communicate with each other via calls to MPI functions
- Typically, each process executes in its own address space (local memory)
  - Distributed memory machines
  - SMP machines
- A process can be sequential or multi-threaded
  - MPI does not specify the initial allocation of processes

MPI: Communicator and Rank

- Communicator
  - An ordered set of processes, either system or user defined
  - Default communicator: `MPI_COMM_WORLD`
  - Use function `MPI_COMM_SIZE` to determine how many processes there are in the communicator
- Rank
  - Your process number within a communicator
  - Used for actual sends and receives
  - Use function `MPI_COMM_RANK` to determine the process rank within a communicator

MPI: Files and Functions

- **MPI header files**

Fortran

```
include 'mpif.h'
use MPI
```

C / C++

```
#include <mpi.h>
```

- MPI function format

Fortran

```
CALL MPI_XXXXX(parameter, ... , IERROR)
```

C / C++

```
error = MPI_Xxxxx(parameter, ...);
```

MPI: Startup

- **MPI Initialization**

Fortran

```
CALL MPI_INIT(IERROR)
```

C / C++

```
int MPI_Init(int *argc, char ***argv);
```

- This routine must be called before any other MPI calls
- It may be called only once (subsequent calls are erroneous)

MPI: Shutdown

- **MPI Termination**

Fortran

```
CALL MPI_FINALIZE(IERROR)
```

C/C++

```
int MPI_Finalize();
```

- Cleans up all MPI state
- No further MPI routines (even `MPI_Init`) may be called

MPI: Startup and Shutdown (continued)

- C/C++ example:

```
#include <mpi.h>
    main(int argc, char *argv[]){
    int error;
        error = MPI_Init(&argc, &argv);
        // test for error == MPI_SUCCESS

        // your MPI program

        error = MPI_Finalize();
        // test for error == MPI_SUCCESS
    }
```

MPI: Startup and Shutdown (continued)

- Fortran example:

```
PROGRAM EXAMPLE
USE MPI
IMPLICIT NONE
INTEGER :: IERROR
...
CALL MPI_Init (IERROR)
! check for IERROR == MPI_SUCCESS
...
! Program body
...
CALL MPI_Finalize (IERROR)
! check for IERROR == MPI_SUCCESS
...
END PROGRAM EXAMPLE
```

MPI: Fortran MPI Data Types

- The following data types can be used in all Fortran MPI programs:

```
– MPI_INTEGER          INTEGER
– MPI_LOGICAL          LOGICAL
– MPI_REAL         REAL
– MPI_DOUBLE_PRECISION  DOUBLE PRECISION
– MPI_COMPLEX          COMPLEX
– MPI_DOUBLE_COMPLEX    COMPLEX*16 (or COMPLEX*32)
– MPI_INTEGER1         INTEGER*1
– MPI_INTEGER2         INTEGER*2
– MPI_INTEGER4         INTEGER*4
– MPI_INTEGER8         INTEGER*8
– MPI_REAL4        REAL*4
– MPI_REAL8        REAL*8
```

MPI: C MPI Data Types

- The following data types can be used in all C MPI programs:

```
– MPI_CHAR          char
– MPI_BYTE          unsigned char (see the standard)
– MPI_SHORT         short
– MPI_INT             int
– MPI_LONG          long
– MPI_UNSIGNED_CHAR unsigned char
– MPI_UNSIGNED_SHORT   unsigned short
– MPI_UNSIGNED        unsigned int
– MPI_UNSIGNED_LONG    unsigned long
– MPI_FLOAT         float
– MPI_DOUBLE        double
```

MPI: Send and Receive

- SEND – Standard send: a blocking send operation

Fortran

```
INTEGER COUNT,DATATYPE,DEST,TAG,COMM,IERROR
CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
  IERROR)
```

C / C++

```
int MPI_Send(void *buf, int count, MPI_Datatype
datatype,int dest, int tag, MPI_Comm comm)
```

MPI: Send and Receive (continued)

- RECEIVE – Standard receive: a blocking receive operation

Fortran

```
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, IERROR
INTEGER STATUS(MPI_STATUS_SIZE)
CALL MPI_RECV (BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
  STATUS, IERROR)
```

C / C++

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm, MPI_Status *status)
```

MPI: A Simple Fortran MPI Program

```
PROGRAM SIMPLE ! SAMPLE 2-PE MPI CODE

INCLUDE 'mpif.h'
INTEGER, PARAMETER :: N = 1000
INTEGER OTHER_PE
INTEGER SEND, RECV
INTEGER STATUS(MPI_STATUS_SIZE)
REAL, DIMENSION(N) :: RBUF, SBUF

CALL MPI_INIT(IERR)
IF (IERR /= MPI_SUCCESS) STOP 'BAD INIT'
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPES, IERR)
IF (IERR /= MPI_SUCCESS) STOP 'BAD SIZE'
CALL MPI_COMM_RANK(MPI_COMM_WORLD, ME, JERR)
IF (JERR /= MPI_SUCCESS) STOP 'BAD RANK'
IF (NPES /=2) THEN
  PRINT*,'MUST RUN WITH 2 PES- EXITING'
  CALL EXIT(2)
ENDIF
```

MPI: A Simple Fortran MPI Program (continued)

```
IF (ME == 0) OTHER_PE = 1
IF (ME == 1) OTHER_PE = 0
DO J = 1, N
 SBUF(J) = J
ENDDO
IF (ME == 0) THEN
 CALL MPI_SEND(SBUF, N, MPI_REAL, OTHER_PE, 99, &
  MPI_COMM_WORLD, SEND)
 IF (SEND /= MPI_SUCCESS) STOP 'BAD SEND ON 0'
 CALL MPI_RECV(RBUF, N, MPI_REAL, OTHER_PE, 99, &
  MPI_COMM_WORLD, STATUS, RECV)
 IF (RECV /= MPI_SUCCESS) STOP 'BAD RECV ON 0'
ELSE ! PE 1
 CALL MPI_RECV(RBUF, N, MPI_REAL, OTHER_PE, 99, &
  MPI_COMM_WORLD, STATUS, RECV)
 IF (RECV /= MPI_SUCCESS) STOP 'BAD RECV ON 1'
  CALL MPI_SEND(SBUF, N, MPI_REAL, OTHER_PE, 99, &
   MPI_COMM_WORLD, SEND)
 IF (SEND /= MPI_SUCCESS) STOP 'BAD SEND ON 1‘
ENDIF
```

MPI: A Simple Fortran MPI Program (continued)

```
CALL MPI_FINALIZE(IERR)
IF (IERR /= MPI_SUCCESS) STOP 'BAD FINALIZE'
IFLAG = 1
DO I = 1, N
 IF (RBUF(I) /= SBUF(I)) THEN
  IFLAG = 0
  PRINT*,'PE ', ME,': RBUF(',I,')=',RBUF(I), &
    ' SHOULD BE ', SBUF(I)
 ENDIF
ENDDO
IF (IFLAG == 1) THEN
 PRINT*,'TEST PASSED ON PE ', ME
ELSE
 PRINT*,'TEST FAILED ON PE ', ME
ENDIF

END PROGRAM SIMPLE
```

MPI: A Simple C MPI Program

```
#include <mpi.h> /* sample 2-PE MPI code */
#define N 1000

main(int argc, char *argv[]) {
 int num_procs;
 int my_proc;
 int init, size, rank, send, recv, final;
 int i, j, other_proc, flag = 1;
 double sbuf[N], rbuf[N];
 MPI_Status recv_status;
/* Initialize MPI */
 if ((init = MPI_Init(&argc, &argv)) != MPI_SUCCESS) {
     printf("bad init\n");
     exit(-2); }
/* Determine the size of the communicator */
 if ((size = MPI_Comm_size(MPI_COMM_WORLD, &num_procs)) != MPI_SUCCESS) {
     printf("bad size\n");
     exit(2); }
```

MPI: A Simple C MPI Program (continued)

```
/* Make sure we run with only 2 processes */
 if (num_procs != 2) {
   printf("must run with 2 processes\n");
   exit(1); }
/* Determine process number */
 if ((rank = MPI_Comm_rank(MPI_COMM_WORLD, &my_proc)) != MPI_SUCCESS) {
   printf("bad size\n");
   exit(1); }
 if (my_proc == 0) other_proc = 1;
 if (my_proc == 1) other_proc = 0;
 for (i = 0; i < N; i++)
   sbuf[i] = i;
```

MPI: A Simple MPI Program (continued)

```
/* Both processes send and receive data */
 if (my_proc == 0) {
  if ((send = MPI_Send(sbuf, N, MPI_DOUBLE, other_proc, 99,
                       MPI_COMM_WORLD)) != MPI_SUCCESS) {
   printf("bad send on %d\n",my_proc);  exit(1); }

  if ((recv = MPI_Recv(rbuf, N, MPI_DOUBLE, other_proc, 98,
                       MPI_COMM_WORLD, &recv_status)) != MPI_SUCCESS) {
   printf("bad recv on %d\n", my_proc);  exit(1);}
 }
 else if (my_proc == 1) {
  if ((recv = MPI_Recv(rbuf, N, MPI_DOUBLE, other_proc, 99,
                       MPI_COMM_WORLD, &recv_status)) != MPI_SUCCESS) {
   printf("bad recv on %d\n", my_proc);  exit(1); }

  if ((send = MPI_Send(sbuf, N, MPI_DOUBLE, other_proc, 98,
                       MPI_COMM_WORLD)) != MPI_SUCCESS) {
   printf("bad send on %d\n",my_proc);  exit(1); }
 }
```

MPI: A Simple C MPI Program (continued)

```
/* Terminate MPI */
 if ((final = MPI_Finalize()) != MPI_SUCCESS) {
   printf("bad finalize \n");
   exit(1);
 }
/* Making sure clean data has been transfered */
 for(j = 0; j < N; j++) {
   if (rbuf[j] != sbuf[j]) {
     flag = 0;
     printf("process %d: rbuf[%d]=%f. Should be %f\n",
       my_proc, j, rbuf[j], sbuf[j]);
   }
 }
 if (flag == 1)
   printf("Test passed on process %d\n", my_proc);
 else
   printf("Test failed on process %d\n", my_proc);
}
```

MPI: Fortran MPI Example 2

```
PROGRAM RING
INCLUDE 'mpif.h'
INTEGER :: RIGHT
INTEGER :: SEND, RECV, TOKEN, OTHER
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: STATUS

CALL MPI_INIT(IERR)
   IF (IERR /= 0) STOP 'BAD INIT'
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPES, IERR)
   IF (IERR /= MPI_SUCCESS) STOP 'BAD SIZE'
CALL MPI_COMM_RANK(MPI_COMM_WORLD, ME, JERR)
   IF (JERR /= MPI_SUCCESS) STOP 'BAD RANK‘

RIGHT = MOD((ME + 1), NPES)
LEFT = MOD((ME + NPES - 1), NPES)
```

MPI: Fortran MPI Example 2 (continued)

```
TOKEN = ME + 1
SUM = 0.0
DO I = 1, NPES
   IF (MOD(ME, 2) == 0) THEN
      ! EVEN PEs
      CALL MPI_SEND(TOKEN, 1, MPI_INTEGER, RIGHT, LFLAG,&
                    MPI_COMM_WORLD, SEND)
      IF (SEND /= MPI_SUCCESS) STOP 'BAD SEND ON EVEN‘

      CALL MPI_RECV(OTHER, 1, MPI_INTEGER, LEFT, LFLAG, &
                    MPI_COMM_WORLD, STATUS, RECV)
      IF (RECV /= MPI_SUCCESS) STOP 'BAD RECV ON EVEN'
   ELSE
```

MPI: Fortran MPI Example 2 (continued)

```
      ! ODD PEs
       CALL MPI_RECV(OTHER, 1, MPI_INTEGER, LEFT, LFLAG, &
                     MPI_COMM_WORLD, STATUS, RECV)
      IF (RECV /= MPI_SUCCESS) STOP 'BAD RECV ON ODD‘

      CALL MPI_SEND(TOKEN, 1, MPI_INTEGER, RIGHT, LFLAG,&
                    MPI_COMM_WORLD, SEND)
      IF (SEND /= MPI_SUCCESS) STOP 'BAD SEND ON ODD'
    ENDIF
    TOKEN = OTHER
    SUM = SUM + TOKEN
ENDDO

PRINT*,'PE ',ME,' TOTAL=',SUM

CALL MPI_FINALIZE(IERR)
END PROGRAM RING
```

MPI: C MPI Example 2

```
#include <mpi.h>

main(int argc, char *argv[]) {
int num_procs;
int my_proc;
int init, size, rank, send, recv, final;
int i, j, other_proc, flag = 1, tag=101;
int right, left;
int token, other, sum = 0.0;
MPI_Status recv_status;
```

MPI: C MPI Example 2 (continued)

```
if ((init = MPI_Init(&argc, &argv)) != MPI_SUCCESS)
  { printf("bad init\n"); exit(2); }

if ((size = MPI_Comm_size(MPI_COMM_WORLD, &num_procs)) != MPI_SUCCESS)
  {printf("bad size\n"); exit(2); }

if ((rank = MPI_Comm_rank(MPI_COMM_WORLD,&my_proc)) != MPI_SUCCESS)
  {printf("bad size\n");exit(2);}

token = my_proc + 1;
right = (my_proc + 1) % num_procs;
left  = (my_proc + num_procs - 1) % num_procs;
```

MPI: C MPI Example 2 (continued)

```
for (i = 0; i < num_procs; i++) {
   if ((my_proc %2) == 0) {
      if ((send = MPI_Send(&token, 1, MPI_INT, right, tag,
           MPI_COMM_WORLD)) != MPI_SUCCESS) {
         printf("bad send on %d\n",my_proc);
         exit(1);
      }

      if ((recv = MPI_Recv(&other, 1, MPI_INT, left, tag,
           MPI_COMM_WORLD, &recv_status)) != MPI_SUCCESS) {
         printf("bad recv on %d\n", my_proc);
         exit(1);
      }
   }
```

MPI: C MPI Example 2 (continued)

```
   else {
      if ((recv = MPI_Recv(&other, 1, MPI_INT, left, tag,
           MPI_COMM_WORLD, &recv_status)) != MPI_SUCCESS) {
         printf("bad recv on %d\n", my_proc);
         exit(1); }

      if ((send = MPI_Send(&token, 1, MPI_INT, right, tag,
           MPI_COMM_WORLD)) != MPI_SUCCESS) {
         printf("bad send on %d\n",my_proc);
         exit(1); }
   }
   token = other;
   sum += token;
}
if ((final = MPI_Finalize()) != MPI_SUCCESS) {
   printf("bad finalize \n"); exit(1); }

printf("PE %d:\tSum= %d\n", my_proc, sum)
}
```

MPI: Send / Receive modes

- MPI blocking sender modes
  - Standard send MPI_SEND
  - Synchronous send MPI_SSEND
  - Buffered send MPI_BSEND
  - Ready send MPI_RSEND
- Non-blocking operations
  - Standard send MPI_ISEND
  - Synchronous send MPI_ISSEND
  - Buffered send MPI_IBSEND
  - Ready send MPI_IRSEND
  - Receive MPI_IRECV

MPI: Modified Fortran MPI Example 2

– Add a few variable declarations

```
INTEGER:: REQUEST
INTEGER, DIMENSION(MPI_STATUS_SIZE)::STATUS
```

– Change the main loop

```
DO I=1, NPES
 CALL MPI_ISEND(TOKEN , 1, MPI_INTEGER, RIGHT , LFLAG, &
     MPI_COMM_WORLD , REQUEST, SEND)
 CALL MPI_RECV(OTHER, 1, MPI_INTEGER, LEFT, LFLAG, &
     MPI_COMM_WORLD , STATUS, RECV)
 CALL MPI_WAIT (REQUEST, STATUS, IERROR)
```

MPI: Modified C MPI Example 2

- Use the non-blocking send, to modify the previous C language example:
  - Add a few variable declarations

```
MPI_Status send_status;
MPI_Request request;
```

  - Change the main loop

```
for( i = 0; i < size; i++) {
   MPI_Isend(&token, 1, MPI_INT, right, tag,
      MPI_COMM_WORLD, &request);
   MPI_Recv(&other, 1, MPI_INT, left, tag,
      MPI_COMM_WORLD, &recv_status);
   MPI_Wait(&request, &send_status);
```

MPI: Collective Communication

- Communication that involves a group of processes
  - Barrier synchronization
  - Broadcast
  - Global reduction operations (e.g., sum, min, max, user-defined)
  - Gather/scatter operations and their variants
  - Combined reduction and scatter
  - Scan (prefix) operations

MPI: Barrier Synchronization

- C/C++

```
int MPI_Barrier (MPI_Comm comm)
```

- Fortran

```
INTEGER::COMM,IERROR
CALL MPI_BARRIER (COMM, IERROR)
```

  - The calling process blocks until all group members have called the barrier
  - Useful for synchronization among processes

MPI: Broadcast a Message

- C/C++

```
int MPI_Bcast (void* buf, int count, MPI_Datatype
               datatype, int root, MPI_Comm comm)
```

- Fortran

```
INTEGER::COUNT, DATATYPE, ROOT, COMM, IERROR
<type>::BUF(*)
CALL MPI_BCAST(BUF, COUNT, DATATYPE,
               ROOT, COMM, IERROR)
```

  - Broadcasts a message from the process with rank ROOT to all processes of the group

MPI: Global Reduction Operations

- C/C++

```
int MPI_Reduce (void* sendbuf, void* recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op,
                int root, MPI_Comm comm)
```

- Fortran

```
INTEGER::COUNT,DATATYPE,OP ROOT,COMM,IERROR
<type>::SENDBUF(*), RECVBUF(*)
CALL MPI_REDUCE (SENDBUF, RECBUF, COUNT, DATATYPE, OP,
                 ROOT, COMM, IERROR)
```

– Performs a global reduce operation (predefined or user-defined)

MPI: Global Reduction Operations (continued)

- Predefined operations are:
  - `MPI_MAX` (maximum)
  - `MPI_MIN` (minimum)
  - `MPI_SUM` (sum)
  - `MPI_PROD` (product)
  - `MPI_LAND` (logical and)
  - `MPI_LOR` (logical or)
  - `MPI_BOR` (bit-wise or)
  - `MPI_LXOR` (logical xor)
  - `MPI_BXOR` (bit-wise xo)
  - `MPI_MAXLOC` (max value & location)
  - `MPI_MINLOC` (min value & location)

Interoperability of MPI

- MPI is compatible with:
  - The other distributed memory models of SHMEM, UPC, and Co-array Fortran (CAF)
  - The shared memory model of Pthreads and OpenMP

Hybrid Programming: OpenMP and MPI

Intel MPI has a thread-safe version of the MPI Library.

Use `-mt_mpt` when linking to use the thread-safe version.

If linking with `-openmp`, or `-parallel`, the thread-safe version is provided by default.

Hybrid Programming: OpenMP and MPI

There are three versions of the thread-safe library:

- **Funnelled**:
  The process can be multi-threaded, but only the main thread will make MPI calls

- **Serialized**:
  The process can be multi-threaded, and multiple threads can make MPI calls, but only one at a time. MPI calls are not made concurrently from two distinct threads.

- **Multiple**:
  Multiple threads can call MPI.

The default is funnelled.

Hybrid Programming: OpenMP and MPI

Use `MPI_Init_Thread()` instead of `MPI_Init()` to specify the thread support required by the MPI library:

C:

```
#include <mpi.h>
int MPI_Init_thread ( int *argc, char ***argv, int required, int *provided );
int MPI_Query_thread ( int *provided );
int MPI_Is_thread_main ( int *flag );
```

Fortran:

```
INCLUDE "mpif.h" (or USE MPI)
INTEGER required, provided, ierror
LOGICAL flag
CALL MPI_INIT_THREAD(required, provided, ierror)
CALL MPI_QUERY_THREAD(provided, ierror)
CALL MPI_IS_THREAD_MAIN(flag, ierror)
```

The level of thread support (`required`) is specified by the values:

`MPI_THREAD_FUNNELED, MPI_THREAD_SERIALIZED, MPI_THREAD_MULTIPLE`

MPI Performance Optimization

- Useful optimizations:
  - Use a few large messages rather than many small messages
  - Use nonblocking sends/receives where useful local work can be interleaved
  - Avoid use of unnecessary barrier functions
  - Use the optimized collective functions rather than point-to-point communications
  - Use meaningful tag values on all messages
  - Receive messages by using specific PEs (rather than `MPI_ANY_SOURCE`)

Compiling and Running MPI Codes using Intel MPI

- The Intel MPI Library is a multi-fabric message passing library that implements the Message Passing Interface, v2 (MPI-2) specification.

- It supports the selection of different interconnection fabrics at run time.

Compiling and Running MPI Codes using Intel MPI

Using the Intel MPI Library involves the following steps:

- Compile and link the application using **`mpiicc`**, **`mpiicpc`**, or **`mpiifort`**
- Set up the MPD daemons using **`mpdboot`**
- Select network fabric or device setting the environment variable, **`I_MPI_DEVICE`**
- Run the MPI program using **`mpiexec`**
- Shutdown the MPD daemons using **`mpdallexit`**

OR

- Compile and link the application using **`mpiicc`**, **`mpiicpc`**, or **`mpiifort`**
- Select network fabric or device setting the environment variable, **`I_MPI_DEVICE`**
- Run the MPI program using **`mpirun`**

PBS job submission script for Intel MPI Codes

```
#!/usr/bin/tcsh

#PBS -l walltime=60:00
#PBS -l select=2:ncpus=8:mem=8g:mpiprocs=8
#PBS -j oe

cd $PBS_O_WORKDIR

setenv I_MPI_DEBUG 3
setenv GMON_OUT_PREFIX gmon.out

### Determine how many CPUs, node and processes per node we are using.
cat $PBS_NODEFILE > mpd.hosts
set NP=`cat mpd.hosts | wc -l`
set NN=`sort -u mpd.hosts | wc -l`
set PPN=`expr $NP / $NN`

setenv I_MPI_DEVICE rdssm
setenv I_MPI_DAPL_PROVIDER OpenIB-cma
setenv I_MPI_PIN_PROCS 0,2,4,6,1,3,5,7
setenv I_MPI_FAST_COLLECTIVES enable

mpdboot -n $NN -r ssh -f mpd.hosts
mpdtrace
mpiexec -perhost $PPN -np $NP test1
mpdallexit
```

Documentation for Intel MPI

Documentation for the Intel MPI is included with the installation:

```
> which mpirun
/sw/sdev/intel/impi/3.2.0.011/bin64/mpirun
> ls /sw/sdev/intel/impi/3.2.0.011/doc
Getting_Started.pdf
INSTALL.html
Reference_Manual.pdf
Release_Notes.txt
```