Introduction to MPI Programming
Outline

- Introduction and concepts
- Introduction to MPI
- Point to Point Communication
- Non-blocking Operations
- MPI 3.0 standard
- Intel® MPI
- Cluster Exploration
- Intel® Trace Analyzer and Collector (ITAC)
Introduction

Execution

Serial

Parallel

ILP

DLP

TLP

CLP

LP – Level Parallelism
I – Instruction | D – Data | T – Thread | C - Cluster
Introduction

Memory Architectures

- Shared Memory Architecture
- Distributed Memory Architecture
HW paradigm - OMP v/s MPI

Shared memory – TLP – Multithreading - OMP

Distributed memory – CLP - MPI

Hybrid computing framework – MPI with OMP
MPI - Paradigm

Pure distributed memory systems

Hybrid memory systems
The SPMD Model

Abstractions make programming and understanding easier

- MIMD, SIMD, SPMD,….

**Single Program Multiple Data (SPMD)**

- Multiple instances of a single program, working on multiple parts of the data
- Each program instance receives a unique ID, which can be used for control flow and for data exchange.

```java
if (myID == specificID) {
    do something
}
else {
    do something different
}
```
SPMD Program Lifecycle

Source Code → Compile & Link → Executable → SPMD Launch → Parallel Execution

SPMD Job

OS Process

data

Result
The Process Space

A process is an in-memory instance of an executable

- Executable code, e.g. binary/machine code produced by a compiler
- Memory context (heap, data)
- One or more threads of execution
- Operating system context (e.g. signal, I/O handle, etc.)
- PID

Process isolation and memory protection

- Process cannot interoperate with other processes or access other contexts without the help of operating system
- No direct inter-processes data exchange
- No direct inter-process synchronization
Outline

• Introduction and concepts
• Introduction to MPI
• MPI standard
• Point to Point Communication
• Non-blocking Operations
• Intel® MPI
• Cluster Exploration
• Intel® Trace Analyzer and Collector (ITAC)
What is MPI – Message Passing Interface?

Common misconceptions

1. MPI is a library
2. MPI parallelizes your application
3. MPI is for distributed memory systems only

• MPI is an open standard library interface for message passing distributed memory systems.
• MPI primarily addresses the *message-passing parallel programming model:* data is moved from the address space of one process to that of another process through cooperative operations on each process
MPI Versions

<table>
<thead>
<tr>
<th>MPI Versions</th>
<th>Release dates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>May 1994</td>
</tr>
<tr>
<td>1.1</td>
<td>June 1995</td>
</tr>
<tr>
<td>1.2</td>
<td>July 1997</td>
</tr>
<tr>
<td>2.0</td>
<td>July 1997</td>
</tr>
<tr>
<td>2.1</td>
<td>June 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>Sept. 2009</td>
</tr>
<tr>
<td>3.0</td>
<td>Sept. 2012</td>
</tr>
<tr>
<td>3.1</td>
<td>June 2015</td>
</tr>
</tbody>
</table>

- Over 430 routines are defined in the MPI-3 standard, which includes the majority of those in MPI-2 and MPI-1.
- However, most MPI programs can be written using a handful of routines.
General structure of MPI program

Startup, initialization, finalization and shutdown – C/C++

1. Inclusion of the MPI header file
   ```
   #include <mpi.h>
   ```

2. Non-coordinated running mode: serial
   - NO MPI function calls allowed with few exceptions
   - All program instances run exactly the same code

3. Initialisation of the MPI environment
   Implicit synchronisation

4. Parallel code, typically computation and communication

5. Termination of the MPI environment
   Internal buffers are flushed

6. Non-coordinated running mode: serial
   - NO MPI function calls allowed afterwards with few exceptions

C/C++

Start-up, initialization, finalization and shutdown

```
int main(int argc, char **argv)
{
    ... some code ...
    MPI_Init(&argc, &argv);

    ... computation & communication ...

    MPI_Finalize();
    ... wrap-up ...
    return 0;
}
```
MPI program structure and routines

- **MPI_INIT(args)**
- **MPI_COMM_RANK(args)**
- **MPI_COMM_SIZE(args)**
- **MPI_SEND(args)**
- **MPI_RECV(args)**
- **MPI_ISEND(args)**
- **MPI_Irecv(args)**
- **MPI_BCAST(args)**
- **MPI_IBCAST(args)**
- **MPI_IBARRIER(args)**
- **MPI_BARRIER(args)**
- **MPI_ABORT(args)**
- **MPI_FINALIZE(args)**

Legend:
- **Environment management**
- **Point to point**
- **Collective**
- **Non-blocking**
MPI – Communicator and ranks

- Processes in an MPI program are initially indistinguishable
- After initialization MPI assigns each process a unique identity – rank
MPI – Communicator and ranks

- Processes in an MPI program are initially indistinguishable
- After initialization MPI assigns each process a unique identity – rank
- Ranks range from 0 up to the number of processes minus 1
General structure of MPI program

- How many processes are there?
- Who am I?

```c
#include <mpi.h>
int main(int argc, char **argv)
{
    // ... some code ...
    MPI_Init(&argc, &argv);
    // ... other code ...
    MPI_Comm_size(MPI_COMM_WORLD, &numberOfProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // ... computation & communication ...
    MPI_Finalize();
    // ... wrapup ...
    return 0;
}
```

1. Obtain the number of processes (ranks) in the MPI program instance
   - E.g. if there are 2 processes running then `numberOfProcs` will contain 2 after the call

2. Obtain the identity of the calling process in the MPI program
   - Note: MPI processes are numbered starting from “0”
   - E.g. if there are 2 processes running then `rank` will be “0” in the first process and “1” in the second process after the call
Initialization

Before any parallel work could be done, MPI has to be initialized:

```c
int MPI_Init (int *pargc, char **pargv)
```

- `pargc` and `pargv` as received by `main` should be provided to `MPI_Init`
- MPI establishes the so called *MPI universe* during the call to `MPI_Init` and it could take quite a long time for large MPI jobs

```fortran
SUBROUTINE MPI_INIT (ierr)
```

- A single argument, which receives the error code of the initialization process
Finalization

After all parallel work is done, MPI has to be finalized:

```c
int MPI_Finalize ()
```

```fortran
SUBROUTINE MPI_FINALIZE (ierr)
```

- All pending communications are first completed
- After the call finishes the calling process is no longer a member of the MPI universe and cannot make further MPI calls

- **Important**: if one or more processes exit without calling MPI_Finalize the behaviour of the MPI program is undefined
- Most MPI implementations terminate the whole MPI job
Checkup on SPMD Requirements

- Provide identification of all participating processes
  - Who am I and who is also working on this problem?

- Provide robust mechanisms to exchange data
  - Whom to send data to?
  - How much data?
  - What kind of data?
  - Has the data arrived?

- Provide a synchronization mechanism
  - Are all processes at the same point in the program execution flow?

- Provide method to launch and control a set of processes
  - How do we start multiple processes and get them to work together?
Outline

• Introduction and concepts
• Introduction to MPI
• **Point to Point Communication**
• Non-blocking Operations
• MPI 3.0 standard
• Intel® MPI
• Cluster Exploration
• Intel® Trace Analyzer and Collector (ITAC)
Explicit message passing

• Recall: the goal is to enable communication between processes that do not share memory

• Required:
  - Send and Receive primitives (operations)
  - Identification of both the sender and the receiver
  - Specification of what has to be send/received
## Sending data

To send a single message:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI_Send</strong></td>
<td>Send data to a single destination</td>
<td><code>void *data, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm</code></td>
</tr>
<tr>
<td></td>
<td><strong>What?</strong></td>
<td><strong>To whom?</strong></td>
</tr>
<tr>
<td></td>
<td><code>data</code>: location in memory of the data to be sent</td>
<td><code>C/C++</code></td>
</tr>
<tr>
<td></td>
<td><code>count</code>: number of data elements to be sent</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>type</code>: MPI datatype of the data elements</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>dest</code>: rank of the receiver</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>tag</code>: additional identification of the message</td>
<td><code>MPI\_Send (DATA, COUNT, TYPE, DEST, TAG, COMM, IERR)</code></td>
</tr>
<tr>
<td></td>
<td>(colour, tag, etc.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>comm</code>: communication context (communicator)</td>
<td></td>
</tr>
</tbody>
</table>

```c
INTEL
```
Receiving data

To receive a single message:

- **data**: location of the receive buffer
- **count**: size of the receive buffer in data elements
- **type**: MPI datatype of the data elements in the receive buffer
- **source**: rank of the sender or the MPI_ANY_SOURCE wildcard
- **tag**: message tag or the MPI_ANY_TAG wildcard
- **comm**: communication context
- **status**: status of the receive operation or MPI_STATUS_IGNORE

**MPIRecv** (void *data, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status)

**MPI_RECV** (DATA, COUNT, TYPE, SRC, TAG, COMM, STATUS, IERR)
MPI datatypes

- MPI provides many predefined datatypes for each language binding
  - Fortran
  - C/C++
  - User defined datatypes

<table>
<thead>
<tr>
<th>MPI data type</th>
<th>C data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>-</td>
</tr>
</tbody>
</table>
MPI datatypes

• **MPI datatypes tell MPI how to:**
  - read actual memory values from the send buffer
  - write actual memory values in the receive buffer
  - convert between machine representations in heterogeneous environments

• **MPI_BYTE is used to send and receive data as-is without any conversion.**

• **Matching datatypes are required on both sides of the connection.**

• **MPI datatypes must match the language data type of the data array.**

• **MPI datatypes are handles and cannot be used to declare variables.**
Return values and error handling

- Almost every C/C++ MPI call returns an integer error code:
  \[ \text{int MPI\_Send}(\ldots) \]

- Error codes indicate the success of the operation:
  \[ \text{C/C++} \quad \text{MPI\_SUCCESS} == \text{MPI\_Send}(\ldots) \]
  \[ \text{Failure indicated by error code different from MPI\_SUCCESS} \]

- If an error occurs, an MPI error handler is called before the operation returns. The default MPI error handler aborts the MPI job!
Message passing: The MPI way

• Message passing in MPI is explicit:

- Rank 0
  - `MPI_Recv(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);`
  - `a^0 = b^1;`

- Rank 1
  - `MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);`

• These two calls transfer the value of the `b` variable in rank 1 into the `a` variable of rank 0

• For now assume that `comm` is fixed as MPI_COMM_WORLD. We will talk about communicators later on.
Checkup on SPMD Requirements

• Provide identification of all participating processes
  ✓ Who am I and who is also working on this problem?

• Provide robust mechanisms to exchange data
  ✓ Whom to send data to?
  ✓ How much data?
  ✓ What kind of data?
  ❖ Has the data arrived?

• Provide a synchronization mechanism
  ▪ Are all processes at the same point in the program execution flow?

• Provide method to launch and control a set of processes
  ▪ How do we start multiple processes and get them to work together?
A full MPI program

#include <mpi.h>
int main(int argc, char **argv)
{
    int numProc, rank, data;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0)
        MPI_Recv(&data, 1, MPI_INT, 1, 0,
                MPI_COMM_WORLD, &status);
    else if (rank == 1)
        MPI_Send(&data, 1, MPI_INT, 0, 0,
                 MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
Message envelope and matching

- Reception of MPI messages is done by matching their envelope.
- Recall: MPI_Send

```
MPI_Send (void *data, int count, MPI_Datatype type, 
          int dest, int tag, MPI_Comm comm)
```

- Message Envelope

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Implicit, Explicit, wildcard possible (MPI_ANY_SOURCE)</td>
</tr>
<tr>
<td>Destination</td>
<td>Explicit, Implicit</td>
</tr>
<tr>
<td>Tag</td>
<td>Explicit, Explicit, wildcard possible (MPI_ANY_TAG)</td>
</tr>
<tr>
<td>Communicator</td>
<td>Explicit, Explicit</td>
</tr>
</tbody>
</table>

- Recall: MPI_Recv

```
MPI_Recv (void *data, int count, MPI_Datatype type, 
          int source, int tag, MPI_Comm comm, MPI_Status *status)
```
Message envelope and matching

- Reception of MPI messages is also dependent on the data.

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send</td>
<td>(void *data, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)</td>
<td>C/C++</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>(void *data, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status)</td>
<td>C/C++</td>
</tr>
</tbody>
</table>

- The standard expects datatypes at both ends to be congruent

- For a receive to complete a matching send has to be posted and v.v.
- Beware: sends and receives are atomic w.r.t. the message
Combing Send and Receive

MPI_Sendrecv (void *senddata, int sendcount, MPI_Datatype sendtype, int dest,
   int sendtag, void *recvdata, int recvcount, int source)

- Guaranteed to not cause deadlocks with matching peers

<table>
<thead>
<tr>
<th></th>
<th>Send</th>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>senddata</td>
<td>recvdata</td>
</tr>
<tr>
<td>Count</td>
<td>sendcount</td>
<td>recvcount</td>
</tr>
<tr>
<td>Type</td>
<td>sendtype</td>
<td>recvtype</td>
</tr>
<tr>
<td>Dest</td>
<td>dest</td>
<td>-</td>
</tr>
<tr>
<td>Source</td>
<td>-</td>
<td>source</td>
</tr>
<tr>
<td>Tag</td>
<td>sendtag</td>
<td>recvtag</td>
</tr>
<tr>
<td>Comm</td>
<td>comm</td>
<td>comm</td>
</tr>
<tr>
<td>Status</td>
<td>-</td>
<td>status</td>
</tr>
</tbody>
</table>
Combining Send and Receive

MPI_Sendrecv (void *senddata, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvdata, int recvcount,

• First sends one message and then receives one message.
• Send and receive buffers must not overlap!

MPI_Sendrecv_replace (void *data, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

• First sends a message to dest, then receives a message from source, using the same memory location, elements count and datatype for both operations.
Outline

• Introduction and concepts
• Introduction to MPI
• Point to Point Communication
• Non-blocking Operations
• MPI 3.0 standard
• Intel® MPI
• Cluster Exploration
• Intel® Trace Analyzer and Collector (ITAC)
Non-blocking Send and Receive

- **Initiation of non-blocking send and receive operations:**
  
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`MPI_Isend (void *data, int count, MPI_Datatype data_type,</td>
<td>Initiate non-blocking send operation.</td>
</tr>
<tr>
<td>int dest, int tag, MPI_Comm comm, MPI_Request *request)`</td>
<td>Request set to handle of initiated operation on success.</td>
</tr>
<tr>
<td>`MPI_Irecv (void *data, int count, MPI_Datatype data_type,</td>
<td>Initiate non-blocking receive operation.</td>
</tr>
<tr>
<td>int source, int tag, MPI_Comm comm, MPI_Request *request)`</td>
<td>Request set to handle of initiated operation on success.</td>
</tr>
</tbody>
</table>
  
  - **request:** on success set to the handle of the initiated operation

- **Blocking wait for the completion of an asynchronous operation:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_Wait (MPI_Request *request, MPI_Status *status)</code></td>
<td>WaitFor completion of an asynchronous operation.</td>
</tr>
</tbody>
</table>
  
  - **request:** handle for an active asynchronous operation freed and set to MPI_REQUEST_NULL on successful return
  - **status:** status of the completed operation
Non-blocking Operations

• Blocking send (w/o buffering) and receive operate as follows

Sender

Program

MPI_Send

Data must remain constant

First message part

Intermediate message part

Intermediate message part

Last message part

Receiver

Program

MPI_Recv

Data must not be used
Non-blocking Operations

- Equivalent with non-blocking send and receive operations:

  Sender:
  - Program
  - MPI_Isend
  - MPI_Wait
  - Data must remain constant

  Receiver:
  - Program
  - MPI_Irecv
  - MPI_Wait
  - Data must not be used

  Magic:
  - First message part
  - Intermediate message part
  - Intermediate message part
  - Last message part
Overlapping Computation and Communication

- Other work can be done while data is being communicated:

  ![Diagram showing Overlapping Computation and Communication](image)
Testing on non-blocking requests

- Test if given non-blocking operation has completed:

  \[
  \text{MPI\_Test} \left( \text{MPI\_Request} \ast \text{request}, \text{int} \ast \text{flag}, \text{MPI\_Status} \ast \text{status} \right)
  \]
  
  → **flag**: true once the operation has completed, otherwise false
  
  → **status**: status of the operation (if completed), only set if flag is true

  Upon completion of the operation (i.e. when flag is true) the operation is freed and the request handle is set to MPI\_REQUEST\_NULL

- If called with a null request (MPI\_REQUEST\_NULL):

  → **MPI\_Wait** returns immediately with an empty status

  → **MPI\_Test** sets flag to true and returns an empty status
Utility Functions

- **Attempt to abort all MPI processes in a given communicator:**

  \[
  \text{MPI\_Abort (MPI\_Comm comm, int errorcode)}
  \]
  
  → `errorcode`: is returned to the OS if supported by the implementation

- **Portable timer function:**

  \[
  \text{double MPI\_Wtime ()}
  \]
  
  → Returns the real time that has elapsed since an unspecified (but fixed between successive invocations) point in the past

- **Obtain a string ID of the processor:**

  \[
  \text{MPI\_Get\_processor\_name (char *name, int *resultlen)}
  \]
  
  → `name`: buffer of at least MPI\_MAX\_PROCESSOR\_NAME characters
  
  → `resultlen`: length of the returned processor ID (w/o the ‘\’0’ terminator)
MPI – Sample code

Character-Proc mapping for the MPI sample program
program thesis_sample_mpi
implicit none
include 'mpif.h'

integer,parameter :: charlen=20,msglen=11
integer :: myrank,numProcs,ierr,unitno,tag,i
integer :: status(MPI_STATUS_SIZE),errorcode
integer :: source,dest
character(charlen) :: myrank_char,my_character,i_char
character(charlen) :: base_name,proc_specific_name,ext
character(LEN=1) :: result_basket(msglen),msg

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numProcs, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
if (myrank.eq.0) then
  write(*,*)'Starting program'
endif !myrank
if (numProcs.ne.msglen+1) then
  write(*,*)'Incorrect number of procs used!'
  write(*,*)'Rerun with 12 procs'
call MPI_ABORT(MPI_COMM_WORLD,errorcode,ierr)
endif !numProcs
write(myrank_char,'(I')')myrank+1
myrank_char = adjustl(myrank_char)
! Every proc reads a suitable data file.
dest = numProcs - 1
if (myrank.ne.dest) then
  base_name = 'sample_mpi'
ext = '.dat'
  proc_specific_name = trim(base_name)///'_'/&
                    trim(myrank_char)//trim(ext)
  unitno = 10 + myrank + (1*numProcs)
  open(unitno, file=proc_specific_name, status='old')
  read(unitno,*)my_character
endif !myrank
if (myrank.eq.0) then
  result_basket(:) = 'X'
  write(*,*)'result_basket=',result_basket
endif !myrank
call MPI_BCAST(result_basket,msglen,MPI_CHARACTER,0, &
               MPI_COMM_WORLD,ierr)
msg = trim(my_character)
if (myrank.eq.dest) then
  do i = 1,numProcs-1
MPI – Sample code – contd.

! Receive message
  call MPI_RECV(result_basket(i), 1, MPI_CHARACTER, &
               i-1, 1, MPI_COMM_WORLD, status, ierr)
  write(i_char,'(I)')i
  i_char = adjustl(i_char)
  write(*,*)'[P12]','ReceivedMessageFrom[P',trim(i_char),',']'
  ! Track progress
  write(*,*)'result_basket=',result_basket
enddo !i
endif !myrank
if (myrank.ne.dest)then
  ! Send message
  call MPI_SEND(trim(msg), 1, MPI_CHARACTER, dest, 1, &
                 MPI_COMM_WORLD, ierr)
  write(*,*)'[P',trim(myrank_char),','] sent message:',' &
          trim(msg),'
          to [P12]'
endif
! Synchronization
  call MPI_BARRIER(MPI_COMM_WORLD,ierr)
if (myrank.eq.0)then
  write(*,*)'Ending program'
endif !myrank
  call MPI_FINALIZE(ierr)
end program thesis_sample_mpi
MPI – Progressive reconstruction

Starting program
result_basket=XXXXXXXXXXXX
[P1] sent message:H to [P12]
[P2] sent message:E to [P12]
[P3] sent message:L to [P12]
[P4] sent message:L to [P12]
[P5] sent message:O to [P12]
[P6] sent message:_ to [P12]
[P7] sent message:W to [P12]
[P8] sent message:O to [P12]
[P9] sent message:R to [P12]
[P10] sent message:L to [P12]
[P11] sent message:D to [P12]
[P12] received message from [P1]
result_basket=HELXXXXXXXX

result_basket=HELXXXXXXXX
[P12] received message from [P4]
result_basket=HELXXXXXXXX

[P12] received message from [P5]
result_basket=HELLOXXXXX

[P12] received message from [P6]
result_basket=HELLOXXXXX

[P12] received message from [P7]
result_basket=HELLO_WXXX

[P12] received message from [P8]
result_basket=HELLO_WXXX

[P12] received message from [P9]
result_basket=HELLO_WOXXX

[P12] received message from [P10]
result_basket=HELLO_WORXX

[P12] received message from [P10]
result_basket=HELLO_WORLX

[P12] received message from [P11]
result_basket=HELLO_WORLD

[P12] received message from [P3]
result_basket=HELLO_WORLD

Ending program
Compile and Link Commands

Using Intel compilers

```bash
mpiicc, mpiicpc, mpiifort
```

Using GNU* compilers (same underlying Intel® MPI Library)

```bash
mpicc, mpicxx, mpif77, ...
```

Ease of use

- Commands find the Intel® MPI Library include files automatically
- Commands link the Intel® MPI libraries automatically

Commands use compilers from PATH (or selected through options), not hard-wired

Example, compile using the Intel® Fortran Compiler

```bash
mpiifort -o testf test.f
```
Execution Command

All-inclusive

- Most common usage scenario
  - Convenient
  - Uses new Hydra process manager by default
  - Better for jobs in batch system
    “In-session” mode: mpirun acquires the list of nodes from the batch system

Example:
- Run the test program

```bash
$ mpirun -f hosts.file -n 2 ./testc
Hello world: rank 0 of 2 running on node1
Hello world: rank 1 of 2 running on node1
```
Process Placement

Simple process placement

\[
\text{mpirun [-perhost #ppn] \text{-n} #\text{procs executable}}
\]

Exact process placement using Argument Sets

\[
\text{mpirun \text{-n} #p1 \text{-host node1 exe1 : -n} #p2 \text{-host node2 exe2}}
\]

- Argument Set (separated by “:”) defines a set of “local” options
  - Local options apply only to current Argument Set
  - Global options apply to all Argument Sets

Exact process placement with a Configuration File

\[
\text{\$ cat theconfigfile} \\
\text{-n #p1 \text{-host node1 exe1}} \\
\text{-n #p2 \text{-host node2 exe2}} \\
\text{# -n #p3 \text{-host dead_node3 exe3}} \\
\text{-n #p4 \text{-host node4 exe4}} \\
\text{\$ mpirun \text{-configfile theconfigfile}}
\]
Outline

• Introduction and concepts
• Introduction to MPI
• Point to Point Communication
• Non-blocking Operations
• MPI 3.0 standard
• Intel® MPI
• Cluster Exploration
• Intel® Trace Analyzer and Collector (ITAC)
MPI 3.0

• MPI 3.0 is an extension of MPI 2.2.
• MPI 3.0 is a major update to the MPI standard.
• New features (3.0) include nonblocking collectives, one sided communication operations and Fortran 2008 bindings.
• MPI 3.1 is a minor update.
• New features (3.1) include nonblocking collective I/O routines, routine to manipulate MPI_Aint values in a portable manner, and routines to get the index value by name for MPI_T_performance and control variables.
MPI-3: Non-blocking Collectives (NBC)

• New Non-blocking versions of collective operations in MPI-3
  • MPI_IBcast, MPI_IBarrier, MPI_IGather, MPI_IAlltoall, etc.
  • more than one non-blocking collective operation can be active on a given communicator
• Pseudo-code for a non-blocking reduction (from Optimizing a Conjugate Gradient Solver with NBC operations, Torsten Hoefler et al.)

```c
MPI_Request req;
int sbuf1[SIZE], rbuf1[SIZE];

/* start non-blocking allreduce of sbuf1 */

MPI_Iallreduce(sbuf1, rbuf1, SIZE, MPI_INT, MPI_SUM, MPI_COMM_WORLD, &req);

MPI_Wait(&req, &stat);
```
Outline

• Introduction and concepts
• Introduction to MPI
• Point to Point Communication
• Non-blocking Operations
• MPI 3.0 standard
• Intel® MPI
• Cluster Exploration
• Intel® Trace Analyzer and Collector (ITAC)
## Value Proposition

<table>
<thead>
<tr>
<th>What</th>
<th>Intel’s High Performance MPI Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why</td>
<td>Scale Performance – Tuned for Latest Intel Architectures</td>
</tr>
<tr>
<td></td>
<td>Scale Forward – Multicore and Manycore Ready</td>
</tr>
<tr>
<td></td>
<td>Scale Efficiently – Flexible Fabric Selection &amp; Compatibility</td>
</tr>
<tr>
<td>How</td>
<td>Standards Based – Built on Open Source MPICH Implementation</td>
</tr>
<tr>
<td></td>
<td>Sustained Scalability – Tuning for Low Latencies, High Bandwidth &amp; Increased Processes</td>
</tr>
<tr>
<td></td>
<td>Multi Fabric Support – Supports Popular High Performance Networking Fabrics</td>
</tr>
</tbody>
</table>
Intel® MPI Library Overview

Optimized MPI application performance
- Application-specific tuning
- Automatic tuning

Lower latency and multi-vendor interoperability
- Industry leading latency
- Performance optimized support for the latest OFED capabilities through DAPL 2.x

Faster MPI communication
- Optimized collectives

Sustainable scalability beyond 262K cores
- Native InfiniBand* interface support allows for lower latencies, higher bandwidth, and reduced memory requirements

More robust MPI applications
- Seamless interoperability with Intel® Trace Analyzer and Collector
Superior Performance with Intel® MPI Library 5.1
1792 Processes, 64 nodes (InfiniBand + shared memory), Linux* 64 Relative (Geomean) MPI Latency Benchmarks (Higher is Better)

Up to 5.2X faster on 64 nodes

Configuration: Hardware: CPU: Dual Intel® Xeon E5-2697v3@2.60Ghz; 64 GB RAM. Interconnect: Mellanox Technologies* MT27500 Family [ConnectX®-3].
Software: RHEL 6.3; OFED 3.5-2; Intel® C/C++ Compiler XE 15.0.3; Intel® MPI Library 5.1; Intel® MPI Benchmarks 4.1.

Optimization Notice: Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.
Outline

- Introduction and concepts
- Introduction to MPI
- Point to Point Communication
- Non-blocking Operations
- MPI 3.0 standard
- Intel® MPI
- Cluster Exploration
- Intel® Trace Analyzer and Collector (ITAC)
Cluster Exploration

Intel® Parallel Studio XE Cluster Edition
Objectives of the lesson

• Learn how to analyze cluster components like nodes and interconnects by using Intel® Parallel Studio XE Cluster Edition tools

• Generic (independent of specific program) MPI benchmarking with Intel® MPI Benchmarks (IMB)

• Learn usage of some Intel® MPI Library environment variables for tuning e.g. process pinning
Cluster Exploration Tools

- **cpuinfo**: included in the Intel® MPI Library package
- **Debug level**: raising the debug level of Intel® MPI Library will provide extra information
- **ifconfig etc**: Linux tools for showing available network devices
- **Intel® MPI Benchmarks (IMB)**: Collection of timed MPI tests for generic MPI performance evaluation
- **MPITUNE**: tuning script for automatic determination of optimal setting. Results can be stored and used on demand.
Cluster Node Exploration

- A current cluster node consist of several sockets (1-4) each hosting a multicore CPU
- Multicore CPUs consist of 2-10 compute cores with 2 hardware threads per core
- Intel® Many Integrated Core processors may have up-to 72 cores with up to 4 hardware threads
Cluster Node Exploration: cpuinfo

- Shows important features of a node: number of sockets, cores per socket including hyper-threads and caches
- Part of the Intel® MPI Library distribution
- Reads its data from /proc/cpuinfo and prints it in a more appropriate format
Cluster Node Exploration: cpuinfo

Attention! Numbering of logical processors depends on BIOS!
Using Environment Variables

• Environment variables may be exported inside your shell and automatically propagated to each rank
• Or, they can be specified on the command line for a single run by:

\[
\text{
$\text{mpirun} \ -\text{genv} \ \text{I\_MPI\_DEBUG} \ 4 \ <\text{program.x}>$
}
\]

-genv stands for global environment propagated to all nodes

• It is also possible to define local environments for different nodes:

\[
\text{
$\text{mpirun} \ -\text{env} \ \text{OMP\_NUM\_THREADS} \ 4 \ -n \ 2 \ <\text{program1.x}> \ : \ \backslash$
\text{-env} \ \text{OMP\_NUM\_THREADS} \ 2 \ -n \ 4 \ <\text{program2.x}>$
}
\]

-env defines environment variables locally
Cluster Node Exploration: Debug Info

• Setting the I_MPI_DEBUG environment variable increases the information printed to std_out depending on the non negative integer value specified

• For example, I_MPI_DEBUG=4 prints information about process pinning, used network interfaces and Intel MPI Library environment variables set by the user

• Process pinning is the mapping of MPI ranks to hardware resources like cores, sockets, caches etc. Default pinning strategy of Intel MPI Library may depend on version! To increase performance you should control the pinning especially for hybrid programs (pinning domains)
Cluster Node Exploration: Debug Info

Shared memory data transfer

Rank 0 on first core of first socket: proc 0+8

Rank 4 on first core of second socket: proc 1+9
Cluster Node Exploration: **Pinning**

- Pin the ranks to explicit processors using the environment variable as shown below:

  ```
  $ export I_MPI_PIN_PROCESSOR_LIST=p1,p2,p3,...
  ```

  rank \#n is mapped to logical processor p\_n

Besides explicit mapping of ranks to logical processors as shown, you can also use the predefined settings.
I_MPI_PIN_PROCESSOR_LIST=1-8

First rank on socket #0 and core #0
Second rank on socket #1 and core #1
Cluster **Network** Exploration

- Most current HPC clusters have standard Gigabit Ethernet networks and additionally an InfiniBand* network (other choices are available but not as popular).
- The head node has a public Ethernet connection, and for the communication to the other nodes there is also a private TCP network for the Network File System (NFS) or non-MPI programs.
- InfiniBand* network is used because of its very high bandwidth and low latency.
- Shared memory inside a node can be viewed as an additional network with even higher performance.
Cluster Structure

- IB router
- ETH router
- Head Node: Compile, Edit, Job management
- Internet
- Intra Socket
- Inter Node
- Inter Socket (QPI)
Three Levels of Communication Speed

• Communication speed is not homogeneous:
  • Inter node (Infiniband*, Ethernet, etc)
  • Intra node inter sockets (Quick Path Interconnect QPI)
  • Intra socket

• Two additional levels when using Intel® Xeon Phi™ coprocessor:
  • host Intel® Xeon Phi™ coprocessor communication
  • Inter Intel® Xeon Phi™ coprocessor communication
Selecting Network Fabrics

- Intel® MPI Library automatically uses the best network fabric it can find
- The best fabric is usually InfiniBand* for inter node communication and shared memory for intra node
- The user can select a different fabric using the environment Variable: I_MPI_FABRICS (see labs for possible settings)
Measuring Comm Speed with IMB

The most simple benchmark in IMB is called PingPong: data packages of different size are sent from rank 0 to rank 1 and back:

```
$ mpirun -n 2 IMB-MPI1 pingpong
```
Placing MPI Ranks on a Cluster

• Process placement on a single node was already discussed

• The default strategy for mapping MPI ranks on a cluster tries to balance resources (same number of processes on each socket) and to minimize the distance between adjacent ranks

• A mapping with 2 MPI ranks on different nodes may be enforced by using the flag “–ppn 1”
  
  • PPN stands for Processes Per Node
  
  • Parameter value “1” will place first rank on first node and the second rank on the next node (alternative env. Var.: I_MPI_PERHOST=1)
Measuring 3 Levels of Comm Speed

Inter node communication (e.g. InfiniBand*):

```shell
$ mpirun -ppn 1 -n 2 IMB-MPI1 pingpong
```

Intra node inter socket (QPI):

```shell
$ export I_MPI_PIN_PROCESSOR_LIST=allsocks
$ mpirun -n 2 IMB-MPI1 pingpong
```

Intra node intra socket (between cores on a processor)

```shell
$ export I_MPI_PIN_PROCESSOR_LIST=allcores:grain=1
$ mpirun -n 2 IMB-MPI1 pingpong
```
Three Different Comm Levels
Outline

• Introduction and concepts
• Introduction to MPI
• Point to Point Communication
• Non-blocking Operations
• MPI 3.0 standard
• Intel® MPI
• Cluster Exploration
• Intel® Trace Analyzer and Collector (ITAC)
Intel Trace Analyzer and Collector (ITAC)

• Measuring MPI time by ITAC
• Simulation of run time using an ideal network
• Splitting run time into components (compute, wait,...)
• Analysis of message passing structure
• Detailed Visualization of MPI programs
• Summary
Choice of process grid

Which choice of process grid is optimal? Total grid: 3600x3600

3x3 2D process grid

1200x1200 grid points per process

1x9 1D process grid

3600x400 grid points per process
Measuring MPI times with ITAC

This Chart shows up automatically after clicking Continue on the start screen:

right click -> Ungroup MPI

shows all MPI functions and the Application time == non MPI run time.

Times are accumulated over all ranks

<table>
<thead>
<tr>
<th>Name</th>
<th>TSelf</th>
<th>TSelf/Call</th>
<th>Total</th>
<th>#Calls</th>
<th>TSelf/Call</th>
<th>Total/Proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>All_Processes</td>
<td>1.60887e-3</td>
<td>1.77979e-3</td>
<td>4</td>
<td>62.744</td>
<td>62.744</td>
<td>76.178</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>92.012</td>
<td>92.012</td>
<td>10454</td>
<td>868.07e-6</td>
<td>3.79217</td>
<td>3.79217</td>
</tr>
<tr>
<td>MPI_Sendrecv</td>
<td>77.9551</td>
<td>77.9551</td>
<td>359749</td>
<td>487.252e-6</td>
<td>5.24515</td>
<td>5.24515</td>
</tr>
<tr>
<td>MPI_Alltoall</td>
<td>73.262</td>
<td>73.262</td>
<td>6976</td>
<td>2.7189e-3</td>
<td>3.08269</td>
<td>3.08269</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>19.3109</td>
<td>19.3109</td>
<td>30081</td>
<td>570.809e-3</td>
<td>0.804.619-3</td>
<td>0.804.619-3</td>
</tr>
<tr>
<td>MPI_Comm_split</td>
<td>5.22849</td>
<td>5.22849</td>
<td>408</td>
<td>12.807e-3</td>
<td>317.727-3</td>
<td>317.727-3</td>
</tr>
<tr>
<td>MPI_Scatter</td>
<td>2.0137</td>
<td>2.0137</td>
<td>24</td>
<td>106.66-6</td>
<td>106.66-6</td>
<td>106.66-6</td>
</tr>
<tr>
<td>MPI_Scatterv</td>
<td>1.65045</td>
<td>1.65045</td>
<td>72</td>
<td>52.4512e-3</td>
<td>70.3837-3</td>
<td>70.3837-3</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>1.27829</td>
<td>1.27829</td>
<td>1608</td>
<td>794.519e-3</td>
<td>62.2899-6</td>
<td>62.2899-6</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>1.11112</td>
<td>1.11112</td>
<td>2608</td>
<td>826.61e-6</td>
<td>66.2966e-3</td>
<td>66.2966e-3</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>375.566e-3</td>
<td>375.566e-3</td>
<td>6552</td>
<td>5.71972e-3</td>
<td>15.548e-3</td>
<td>15.548e-3</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>186.662e-2</td>
<td>186.662e-2</td>
<td>68322</td>
<td>2.8848e-5</td>
<td>7.7692e-3</td>
<td>7.7692e-3</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>48.955e-3</td>
<td>48.955e-3</td>
<td>252</td>
<td>194.266e-3</td>
<td>2.0397e-2</td>
<td>2.0397e-2</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>35.71e-3</td>
<td>35.71e-3</td>
<td>47</td>
<td>748.358e-3</td>
<td>1.5479e-2</td>
<td>1.5479e-2</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>22.030e-3</td>
<td>22.030e-3</td>
<td>29</td>
<td>916.79e-3</td>
<td>3.178e-2</td>
<td>3.178e-2</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>1.296e-4</td>
<td>1.296e-4</td>
<td>384</td>
<td>8.817e-5</td>
<td>2.313e-2</td>
<td>2.313e-2</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>1.296e-3</td>
<td>1.296e-3</td>
<td>660</td>
<td>3.6222e-5</td>
<td>5.416e-3</td>
<td>5.416e-3</td>
</tr>
<tr>
<td>MPI_Comm_free</td>
<td>1.296e-4</td>
<td>1.296e-4</td>
<td>41</td>
<td>1.104e-5</td>
<td>2.620e-2</td>
<td>2.620e-2</td>
</tr>
<tr>
<td>MPI_Type_commit</td>
<td>70e-6</td>
<td>70e-6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Type_create</td>
<td>50e-6</td>
<td>50e-6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Type_contiguous</td>
<td>50e-6</td>
<td>50e-6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Two additional columns showing timings per process: right click ➔ Function Profile Settings
ITAC Agenda

Performing a scaling analysis supported by ITAC

Simulation of run time using an ideal network

Splitting run time into components (compute, wait,...)

Analysis of message passing structure

Detailed Visualization of MPI programs

Summary
Algorithm and Network evaluation

ITAC shows timing of all MPI routines used by a program

The timing of MPI routines may be due to network transfer times caused by interconnect bandwidth limitations

The other possibility are waiting times caused by the algorithm: load imbalance or dependencies
A simple Network Model

The most simple network model defines:

- Latency $L$ = transfer time for 0 byte message
- Bandwidth $BW$ = transfer rate for (asymptotically) large messages
- Message Volume $V$ = data amount sent

The transfer time is:

$$T_{\text{trans}}[V] = L + V/BW$$
ITAC: Ideal Network Simulator

It is extremely complicated to simulate a realistic network!

An extreme case – the ideal network – may be simulated by setting all transfer times to 0. This would mean \( L = 0 \) and \( BW = \infty \) for the simple model.

ITAC offers an ideal network simulation with transfer times set to zero. Compute times (non MPI) will stay the same.

An existing real trace file is used as basis for the simulation.
ITAC: Ideal Network Simulator

With a perfectly balanced algorithm the total MPI time will be vanishing in the ideal case

In most real cases the MPI time will just shrink but not vanish

The remaining part is due to waiting time e.g. when the receiver is starting to receive before the sender is ready to send

Start simulator with: Advanced ➔ Idealization
Simulation details

Test cases for simulation are the 16 nodes configurations: 24x16, 1x384, 384x1

Name for idealized trace file gets additional “ideal”

Press start to continue
Waiting time due to dependencies – Event timeline

MPI_Recv is pure waiting time inside an ideal trace file

MPI_Send time shrinks to 0

MPI_Recv must wait on MPI_Send call
ITAC Agenda

Performing a scaling analysis supported by ITAC
Simulation of run time using an ideal network
**Splitting run time into components (compute, wait,...)**
Analysis of message passing structure
Detailed Visualization of MPI programs
Summary
The simulated MPI time for the ideal network may be regarded as the waiting time $T_{\text{wait}}$ due to imbalance and dependencies:

$$T_{\text{mpi}} = T_{\text{transfer}} + T_{\text{wait}}$$

After generation of an ideal trace file the result can be displayed in the Imbalance Diagram:

Advanced $\rightarrow$ Application Imbalance Diagram
Imbalance diagram – 16 Nodes 24x16

Interconnect time ($T_{\text{transfer,acc}}$)
move mouse over bar: 3.89 [sec]

Imbalance ($T_{\text{wait,acc}}$):
0.714621 [sec]

Application($T_{\text{comp,acc}}$):
7.5545 [sec]
contains some artificial startup time
ITAC Agenda

Performing a scaling analysis supported by ITAC
Simulation of run time using an ideal network
Splitting run time into components (compute, wait,...)
Analysis of message passing structure
Detailed Visualization of MPI programs
Summary
Message Passing Profile: 24x16 grid

Use slider for changing the size of cells or:
Message Profile Settings → Automatic Cell Size

Messages from P1 to P2

From P16 to P0
Total Volume: 2D vs. 1D distribution

2D Distribution with alternating Inter Node Volume

Intra Node

1D Distribution with high Intra Node Volume

Only 8 of 16 nodes shown!
Average Rate: 2D vs. 1D distribution

Much lower rates compared to the 1D case
Number of mesg.: 2D vs. 1D distribution

16X and 17X more Inter Node messages compared to 1D
ITAC Agenda

Performing a scaling analysis supported by ITAC
Simulation of run time using an ideal network
Splitting run time into components (compute, wait,...)
Analysis of message passing structure

Detailed Visualization of MPI programs

Summary
Quantitative Timeline for 16 nodes

Color menu was used to color MPI_Allreduce

3 Iterations. Each Iteration does 3 boundary exchanges and is finished by an MPI_Allreduce

Height of blue columns is proportional to # ranks inside Application time at the same time
Single Iteration Poisson

Same configuration as used before in the mapping section: 24x16

This is the default mapping!
Comparing ITAC traces

Compare before and after optimization e.g. compare boundary exchange with blocking Send/Receive to non blocking Send/Receive

Further potential comparison scenarios:

Compare ideal to real trace

Compare different number of ranks

Compare different mappings
Comparing ITAC traces - HowTo

Open tab: View → Compare

Open another file for a comparison
Comparison: Boundary Exchange

Trace A: Send/Recv
Trace B: Isend/Irecv/Waitall

Wait Time on P6 has vanished!

MPI Wait Time for this boundary exchange shrunk by a factor 0.689
ITAC Agenda

Performing a scaling analysis supported by ITAC
Simulation of run time using an ideal network
Splitting run time into components (compute, wait,...)
Analysis of message passing structure
Detailed Visualization of MPI programs

Summary
Summary

MPI is one of the efficient ways to solve problem in distributed environment.

A few methodologies were presented for performing a MPI analysis.

Cluster Exploration helps to have better understanding of topology and allows to have optimal data layout for a given program.

Intel® MPI is has low latency with faster communication with optimized algorithms, also has sustainable scalability beyond 256K cores.

Intel® Trace Analyzer and Collector helps in having better understanding of a MPI application and offers interesting new features like simulation of ideal traces and the computation of transfer and waiting time.
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2014, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804