Technical Report, IDE0735, May 2010

Evaluation of Parallel Programming Standards
For
Embedded High Performance Computing

Master’s Thesis in Computer Systems Engineering

Muggalla James Emmanuel Roy
Garimela Pradeep Kumar

School of Information Science, Computer and Electrical Engineering
Halmstad University
Evaluation of parallel programming standards for embedded high performance computing
Evaluation of parallel programming standards for embedded high performance computing
Preface

This thesis is the final part of the Master’s degree at the University of Halmstad. During the thesis we have had good discussions and help from our supervisors Bertil Svensson and Anders Åhlander. We would like to thank Ruud van der Pas (senior staff engineer, scalable systems group, Sun Microsystems) for providing help in OpenMP. We would also like to thank Dr Christopher Allen for reading our report and giving us critical feedback on the grammatical and spelling errors in the report.

___________________________________________  ______________________________________

Muggalla James Emmanuel Roy                      Garimela Pradeep Kumar

Halmstad University, May 2010.
Evaluation of parallel programming standards for embedded high performance computing
Abstract

The aim of this project is to evaluate parallel programming standards for embedded high performance computing. There is a huge demand for high computational speed and performance in the present radar signal processing, so more processors are needed to get enough performance. One way of getting high performance is by dividing the work on multiple processors. At the same time, it has to get low communication overhead and good speedup. This has been done by using parallel computing languages such as OpenMP and MPI. We use these parallel programming languages on radar signal benchmark which is similar to many tasks in radar signal processing. For running OpenMP, a shared memory system SUNFIRE E2900 is used and for MPI, a SUNFIRE E2900, containing 8 nodes which uses SUN HPC cluster tools v5 is used. The OpenMP program shows pretty good speedup up to 5 processors, there after an increase in communication overhead is observed. MPI has shown low communication overhead at the beginning but got decreases when the numbers of processors were increased. Both OpenMP and MPI show similar aspects, at certain limit as the number of processors are increased there is decreasing trend in efficiency and increase in communication overhead. According to our results, OpenMP is a relatively easy to use program when compared to MPI. When using MPI it is up to the programmer to make explicit calls in order to parallelize.
Abbreviations

- **OPENMP** Open Specification for Multiprocessing
- **MPI** Message Passing Interface
- **FIR** Finite Impulse Response
- **FFT** Fast Fourier Transform
- **CPU** Central Processing Unit
- **SISD** Single Instruction Single Data
- **MISD** Multiple Instruction Single Data
- **VME** Virtual Machine Environment
- **FPGA** Field Programmable Gate Array
- **FLOPS** Floating Point Operations per Second
- **GFIOPS** Giga Floating Point Operations per Second
- **VXS** VisAccess
- **PGI** Portland Group Inc
- **SUN** Stanford University Network
- **MIMD** Multiple Instruction Multiple Data
- **API** Application Program Interface
### List of Figures

1.1 Example of signal processing .......................................................... 13
1.2 Flynn’s Taxonomy ............................................................................. 14
3.1 Shared memory model of parallel computation .................................. 19
3.2 Structure of OpenMP ....................................................................... 20
4.1 Parallel computation with distributed memory .................................. 23
4.2 MPI program structure ................................................................. 24
6.1 OpenMP demo program-Number of processors versus time taken .... 34
7.1 Reading data across rows .................................................................... 39
7.2 Reading data along different memories ............................................. 39
7.3 Process of computation ....................................................................... 39
7.4 Dividing the work on multiple processors and reading data across different memories . . 41
Evaluation of parallel programming standards for embedded high performance computing
Evaluation of parallel programming standards for embedded high performance computing

7.2 MPI Results ........................................................................................................... 53
7.3 CPU Utilization .................................................................................................... 62

8. Conclusions .......................................................................................................... 63

9. References ............................................................................................................ 65

Appendix A Source code .......................................................................................... 67
   A: 1 C++ RADAR signal benchmark ................................................................. 67
   A: 2 C++ Benchmark.cpp .................................................................................. 69
   A: 3 MPI Benchmark ............................................................................................ 72
1. Introduction to research field

The objective of this work is to evaluate parallel programming standards OpenMP and MPI for embedded high performance computing. A major part of the work has been the development of an application with high inherent data parallelism similar to many tasks in radar signal processing.

1.1 Introduction to RADAR Signal Processing

RADAR stands for Radio Detection and Ranging. RADAR systems use radio waves to detect, determine the distance of and map objects such as aircraft, ships and rain. A transmitter emits radio waves which are reflected by the target and detected by a receiver situated, generally in the same location as the transmitter is [1].

Radar was developed in the mid 20th century. Though most radars use microwave frequencies they are not confined to any particular frequency range. There are some radar units that operate on frequencies well below 100megahertz and others that operate in the infrared and above [2]. The radio signal returned is usually very weak, so amplification is needed for processing the data.

Typical uses are [2]:
- Hazardous weather detection, for storm and wind shear warning.
- Navigational aid, such as assisting air-traffic control, avoiding air-to-air collisions and measuring absolute altitude.
- Ground mapping and creating high-resolution terrain maps.
- Reconnaissance and surveillance
- Military applications such as fire control, tactical bombing and precision bombing.

Signal processing in a modern radar system consists of a series of tasks, each comprising very heavy processing. Usage of multiple processors is very essential in the signal processing chain in next-generation systems. In these systems, with phase-controlled antennas, the amount of calculations will be hundreds and thousands, of times larger than in current systems [3].

Here is an example of radar signal processing which consists of a number of steps that include linear operations such as FIR filter, FFT and matrix multiplications. These operations can be done in the following steps shown in figure 1.1.

![Figure 1.1: Shows the example of signal processing.](image)

Input vector is given to F1 block which is a digital beam forming step, will give an output vector Y by multiplying the input vector X with weight matrix W. The output of F1 block is given to next block F2 (pulse compression), which collects all received energy from one target into a single range bin. This can be accomplished by using FIR filters. The output from the step is given to block F3 which is a
Evaluation of parallel programming standards for embedded high performance computing

Doppler filter bank, converting pulse bins to velocity bins by applying FFT. The output is given to block F4 which acts as an envelope detector to eliminate phase component, since phase information is no longer needed.

Finally the output of F4 is given to F5 which is a constant false alarm ratio function, used to reduce the number of false targets in a specified time interval.

Processing the data collected from these radars takes too long a time when we use a single processor. Hence many processors are needed to get enough performance.

**Parallel Computing** - Software that has been written for serial computation will run on a single computer having a single central processing unit (CPU). The problem is broken into several instructions which are executed one after another. Parallel computing is the simultaneous execution of parts of the same task on multiple processors for obtaining results faster. The task is divided into several subtasks and they are distributed onto the processors as series of instructions.

### 1.2 Classification of computer architecture

“Flynn’s taxonomy is a classification of computer architectures, proposed by Michael J. Flynn in 1966. The four classes defined by Flynn are based upon the number of concurrent instruction and data streams available in the architecture as shown in figure 1.2” [4].

<table>
<thead>
<tr>
<th></th>
<th>Single Instruction</th>
<th>Multiple Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

- Single instruction, single data stream (SISD) - It is a serial or non-parallel computer which doesn’t use parallelism either in the instruction or data streams.

- Multiple instructions, single data stream (MISD) – In this procedure many functional units perform different operations on the same data. Pipeline architectures belong to this type.

- Single instruction, multiple data streams (SIMD) - for certain classes of problems, e.g., those known as data-parallel problems this type of architecture is perfectly suited for achieving high processing rates, as the data can be split into many different independent pieces and the multiple processing units can all operate on them at the same time.

- Multiple instruction, multiple data streams (MIMD) In MIMD, each processor can run an independent task [4].
1.3 Some multiprocessors

Simultaneous processing with two or more processors in one computer is called multi processing; two or more computers processing together are called multi computers. When two or more computers are used, they are tied together with a high-speed channel and share the general workload among them.

Modern radar systems require high level processing power and the ability to deal with the increasing high bandwidths of input. Mercury is the industry's standard company in multiprocessor-based signal processing for airborne, ocean-going and land-based radar systems. These systems scale from two to hundreds of processors and are deployed on over 100 platforms. We have taken a few systems which are examples of multicomputers.

9U VME System - MPC7447A PowerPC G4+ computes nodes, up to 280 PowerPCs and 1.8 TFLOPS in a single chassis, 60 fibre optic interfaces for 6 GB/s fibre I/O maximum, and RACE++ switch fabric provides up to 267 MB/s bandwidth per conversation.

Benefit: Real-time embedded multicomputing for the most demanding applications [5].


ImpactRT 3000 Compact Family of Systems - Scales to 72 processors with 778 GFLOPS or 3.1 TOPS, Up to 57.5 GB/s aggregate over the RapidIO switch fabric, up to 36 GB/s fiber I/O, Multiprocessor software environment with algorithm and image processing libraries [7].

In order to program parallel computers or multi-processors, there are different application program interfaces like parallel virtual machine, POSIX, OpenMP and MPI. In this paper we will explain of OpenMP (Open for Multi-Processing) and MPI (Message Passing Interface) for the given program.

1.4 Shared memory system

A shared memory machine has one single address space and thus all the processors communicate with each other using shared variables. Local caching is necessary to get a fast enough system. This introduces cache coherence problems.

1.4.1 Example of shared memory system

SUN FIRE V880, up to 8 Ultrasparc III cu processors running at 1.2 GHz with a maximum of 64GB main memory. It uses Superscalar SPARC 9 architecture. On chip it contains a level 1 (L1) Cache of 64 KB data and 32 KB instruction as well as 8 MB of level 2 (L2) Cache [11].

1.5 Private memory system

A private memory system also referred to as multicomputer typically contains several private address spaces. It means the memory is local to the processor to which it is connected. There is no shared memory in private memory systems.
A cluster of PC’s is a typical example of a private memory system. PC cluster is much less expensive than a shared memory machine and hence many people choose this type of machine. OpenMP code will not run on such a machine since it requires shared memory. Hence, MPI is used [12]. The mercury systems described above are also examples of private memory systems.
Goal and Approach

The goal of this thesis is to evaluate the efficiency of radar signal programs written in parallel computing languages such as OpenMP and MPI.

Therefore we will evaluate the standards in terms of their capability to:

- **Express Data parallelism**: Data parallelism is a form of parallelization of computer code. In the single instruction stream approach (SIMD), data parallelism is achieved when each processor performs the same task on different data. For instance, if we are running code on 2 processors (CPUs X and Y) in a parallel environment and we wish to do a task on some data Z, it is possible to tell the CPU X to do that task on one part of Z and CPU Y on another part of Z simultaneously, thereby reducing the runtime of the execution.

- **Obtain High Processor utilization**: It can be defined as the percentage of time not in the idle task. If we say processor utilization as ‘P’, then P = 100% – (% of time spent in idle task)

- **Minimize Communication overhead**: A measure of the additional workload incurred in a concurrent algorithm due to communication between the nodes of the concurrent processor. If communication is the only source of overhead, then the communication overhead is given by: \((\text{number of processors} \times \text{parallel run time} - \text{sequential run time}) / \text{sequential run time}\) [9].

- **Minimize Latency**: In communication point of view, Latency is the time difference when a request is initiated and actual flow of data taking place [10].

2.1 Approach

To achieve our goals a benchmark is used which is similar to many tasks in RADAR signal processing. For parallelizing the benchmark, we used two different parallel computing languages, namely OpenMP which runs on shared memory machine and MPI on private memory machine. For running OpenMP benchmark we used (SUNFIRE E2900, ULTRASPARC IV dual core processors running 1.2 GHz consisting of 12 processors [8]) and for MPI (Sun Fire E2900 with 12 ULTRASPARC IV Dual core processors running 1.2 GHz, containing 8 nodes using SUN HPC Cluster tools 5 [8]).

To achieve our goals, we start testing benchmark program which is typically used in radar signal processing and run the same program on different (shared memory and private memory) machines.

We test the benchmark program (serial program) and modify the code if necessary, to get good optimization and speedup for a single processor. We distribute the amount of work on multiple processors by parallelizing the code using OpenMP and MPI.

For the given test cases, we check communication overhead and its effects when program is distributed on multiple processors. While running the program we check the CPU utilization (Processor utilization) and how it varies when we increase or decrease the number of processors. Finally we arrive at our results after parallelizing the code in OpenMP and MPI in order to conclude and select which of these languages is better to use and not difficult to implement for applications in radar signal processing.
2.2 Difference between OpenMP and MPI

MPI is mostly used for private memory systems and OpenMP for shared memory systems. A key difference between the shared-memory model and the message passing model (Private memory) is that in the message passing model all processes typically remain active throughout the execution of the program. Whereas in the shared-memory model the number of active threads is one at the program’s start and end and may change dynamically throughout the execution of the program. Often this so-called hybrid model for parallel programming, using both OpenMP and MPI, is used for programming computer clusters [11].

Evaluation of parallel programming standards for embedded high performance computing
3. OpenMP

OpenMP - Open specifications for multi-processing.

OpenMP (shared memory model) is an application program interface for writing multithreaded applications. It provides a portable, scalable model for developers of shared memory parallel applications which make it easy to create multithreaded programs in FORTRAN, C and C++ [12]. The hardware is assumed to be a collection of processors, each with access to the same shared memory. Because they have access to the same memory locations, processors can interact and synchronize with each other through shared variables. Figure 3.1 shows a typical shared memory machine.

![Figure 3.1 Shared memory model of parallel computation.](image)

3.1 Components of OpenMP

OpenMP consists of three major API components

1. Compiler directives
2. Runtime library routines.
3. Environment variables

OpenMP is not meant for distributed memory systems. [12]

3.2 Open MP supporters *

3.2.1 Hardware Vendors:

3.2.2 Software Vendors:
Absoft Corporation, Edinburgh portable Compilers, GENIAS Software GmBH, Myrias Computer Technologies.
Evaluation of parallel programming standards for embedded high performance computing

Application Vendors -


These names of the vendors were taken from [www.openmp.org](http://www.openmp.org) [13].

### 3.3 Programming model

The standard view of parallelism in a shared memory program is fork/join parallelism.

Fork –Join Parallelism -
An Open MP program begins with an initial thread (master thread). When any thread encounters a parallel construct, the master thread creates a team of parallel threads. As shown in figure 3.2. Any number of parallel constructs can be specified in a single program and parallel constructs may be nested [14]. The statements are executed in parallel among the various team threads. When the threads complete the statements in the parallel region, they terminate leaving only the initial thread. In other words when parallel operations are required, the master thread creates or awakens additional threads. The master thread and the created threads work concurrently through the parallel section. At the end of the parallel code the created threads are suspended, and the flow of control returns to the single master thread. Figure 3.2 below shows the structure of OpenMP.

![Figure 3.2: Structure of OpenMP](image)

Figure 3.2: Structure of OpenMP (Master thread starts with thread Id=0 and then it creates slave threads and nested parallelism can be implemented in them).
The ability of the shared-memory model to support incremental parallelization is one of its greatest advantages over the message-passing model. It allows you to profile the execution of a sequential program, sort the program blocks according to how much time they consume. Then consider each block in turn beginning with the most time-consuming, parallelize each block amenable to parallel execution, and stop when the effort required to achieve further performance improvements is not warranted [15]. To be precise Open MP is typically used to parallelize the loops, find the most time consuming loops and split them up between the threads. (Here threads refer to processors)

**Note:** The number of threads should not be more than the number of processors.

We can create any number of threads by setting an environment variable (setenv OMP_DYNAMIC FALSE or export OMP_DYNAMIC=FALSE). By default SUN STUDIO compiler limits the number of threads to 1023.

### 3.4 Compiler directives

Some syntax details for the constructs in OpenMP are compiler directives or pragma's. For C and C++, the pragmas take the form.

```
#pragma omp construct [clause [clause]...]  
```

For FORTRAN the directives take one of the forms:

- C$OMP construct [clause [clause]...]  
- !$OMP construct [clause [clause]...]  
- !$OMP construct [clause [clause]...]  

Directives are case sensitive; otherwise directives will be ignored by the compiler. Clauses on directives may be repeated but only one directive name can be specified per directive. [13]

#### 3.4.1 Parallel construct

This defines a parallel region which has to be executed by the number of threads.

```c
#pragma omp parallel [clause [,] clause...] new-line  
{  
/* region that has to be executed by the threads */  
}
```

Clause can be one of the following.

- `If(scalar-expression)` for example if ‘n’ is less than number of threads do this in parallel or else do the work on single thread.
- **private** (variable-list)
- **firstprivate** (variable-list)
- **default** (shared/none) :-Works only for Fortran and c, default(None) not specified for c++.
- **shared** (variable-list)
- **copyin** (variable-list)
- **reduction** (operator : variable-list)
Evaluation of parallel programming standards for embedded high performance computing

**num_threads** (integer-expression)

At most one if clause can appear on the directive. At most one **num_threads** clause can appear on a directive. The **num_threads** expression must be a positive integer. [13]

### 3.4.2 Determining the number of threads

The number of threads in the parallel region can be determined by using runtime library routines or Environment variables.

Runtime variable = **omp_set_num_threads ()**

Environment variable = **OMP_NUM_THREADS**

Threads are numbered from 0 (master thread) to N-1

### 3.4.3 Dynamic threads

A program which contains multiple parallel regions will use the same number of threads to execute each region. This behaviour can be changed by allowing runtime system to dynamically allow the number of threads [13].

Runtime variable = **omp_set_dynamic ()**

Environment variable = **OMP_DYNAMIC**

### 3.4.4 Nested parallel regions

A parallel region nested within another parallel region results in the creation of a new team. By default each team consists of one thread [13].

Nested parallelism can be implemented by setting **omp_set_nested ()** runtime variable or an environment variable **OMP_NESTED**.

More information about OpenMP and Nested parallelism can be found on [www.openmp.org](http://www.openmp.org).
4. MPI (MESSAGE PASSING INTERFACE)

MPI stands for message passing interface. MPI is not a programming language but instead can be thought of as a library. The model of a computer is one which has a memory, cache and processor. All processors are connected through high speed interconnect. Figure 4.1 shows memory model of Distributed system.

![Memory diagram]

*Figure 4.1: Parallel Computer with Distributed Memory.*

### 4.1 What is message passing?

Multiple separate processes collaborate and communicate with each other by explicitly exchanging (sending and receiving) data, a procedure which is called message passing. These processes have separate address spaces, states, stacks, interrupts and files which are completely different programs. They just communicate by call_send and call_receive in principle. But the programmers are fully responsible for the distribution of data and the work. Principle is very simple but in order to parallelize the algorithm with MPI this may mean a lot of work.

Message passing interface standard was first developed by over 40 organizations, vendors, researchers and users. In MPI there are two standards MPI 1.1 and MPI 2.0, developed during 1994 to 1995.

MPI is a computer communication protocol between a number of nodes running a parallel program on a distributed memory system. The Goal of the message passing interface is to provide convenient, efficient and flexible standard for message passing that will be commonly used for writing message passing programs.

This interface plays a major role in today’s parallel computing applications like signal processing, image analysis and medical applications. In all the above applications we are using multiple processors to execute a particular task. To execute the task between the multiple processors, there should be communication between them to indicate that we use MPI to send messages between the processors [16].
4.2 The basic MPI program structure

4.3 The basics

Fundamental six functions

Initialization of the MPI runtime environment at the very beginning of the program, no MPI function calls before this

MPI_Init (&argc, &argv)

Termination of the MPI runtime environment, no MPI function calls after this

MPI_Finalize ()

Determination of the rank of process in a communication context

MPI_comm_rank (MPI_Comm communicator, int *rank)

Determination of the number of process in a communication context

MPI_Comm_size (MPI_Comm communicator, int *size)
A communicator is a collection of MPI processes that can send messages to each other. A predefined communicator is MPI_COMM_WORLD, which consists of all processes after the program start.

**4.4 The MPI 1.0 standard are**

*Point to point communication:* In this operation the message passing is between two different MPI tasks; one is to perform a send operation and the other is to perform a matching receive operation.

This procedure *sends* data to a process destination.

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

- `Void *buf` – which data
- `Int count` – how much data
- `MPI_datatype` – which datatype
- `Int destination` – to which mpi process
- `Int message_tag` – message identification
- `MPI_comm` – which context

**MPI_Scatter** -

MPI_Scatter makes a subset of the data *send_data* on the processs with rank *root* and sends to each process in the communicator *comm* one of the subsets [17].

```c
int MPI_Scatter (void* send_data, int send_count, MPI_Datatype sendtype, void* receive_data, int receive_count, MPI_Datatype receivetype, int root, MPI_Comm comm)
```

**MPI_Recv** -

This procedure receives data from process source.

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

- `Void *buf` – receive buffer
- `Int count` – buffer size
- `MPI_datatype` – which datatype
- `Int source` – from which mpi process
- `Int message_tag` – message identification
- `MPI_comm` – which context
- `MPI_Status` – message status

*Collective operation:* The process in which all the processors in the communicator involve collective operation. There are a number of types of collective operations, synchronization, data movement and collective computation.
Evaluation of parallel programming standards for embedded high performance computing

- Process groups
- Communication contexts
- Process topologies
- Bindings for FORTRAN 77 and C
- Environmental Management and inquiry
- Profiling interface

Some of the goals of MPI are that it should provide efficient communication between the processors, and is platform independent i.e. it should support on any operating system, programming language. It has to support heterogeneous parallel structures [18].

4.5 Further information about MPI

The MPI forum
http://www.mpi-forum.org/

The MPI home page at Argonne National Lab
http://www-unix.mcs.anl.gov/mpi/

MVAPICH
http://nowlab.cse.ohio-state.edu/projects/mpi-iba/

Open MPI
http://www.open-mpi.org/

Manual pages
man MPI
man mprun
man MPI_Xxx (for all MPI calls)
5. Compilers & tools

Tools are some of the best sources to know what exactly the program is doing. Some of the vendors are providing tools for OpenMP.

5.1 OpenMP

5.1.1 Compilers:

Some of the compilers are provided by SUN, INTEL and PGI.

<table>
<thead>
<tr>
<th>Company</th>
<th>Compilers</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN</td>
<td>Sun Studio Compiler</td>
<td>Sun Performance Analyzer</td>
</tr>
<tr>
<td>INTEL</td>
<td>Intel Compiler</td>
<td>Intel Thread Profiler</td>
</tr>
<tr>
<td>PGI</td>
<td>PGI compiler</td>
<td></td>
</tr>
</tbody>
</table>

Generally in SUN, compilers are invoked with the commands cc, f95, CC.

cc - c program compilation
CC- c++ compilation

Here we would like to give some information about the most commonly used compiler options and flags in SUN STUDIO.

OpenMp program compilation
$ CC -fast -xopenmp -xarch=v9b program.cpp -o program

When compiling the program –fast is always used to boost the performance. $ (bash shell).

-fast is used to get better optimize performance.
-xopenmp is used to enable the OpenMP functions.
-xarch= “a” is for the instruction set.

“a” can be any of the following.
-xarch=v8plusb/v9b 32/64 bit addressing (SPARC)
-xarch=Sse2/amd64 32/64 bit addressing (AMD)

5.1.2 Sun Studio Compilers

Default location of SUN STUDIO compiler.

/opt/SUNWspro/bin
To know which compiler is running;

$cc –v
-xhelp=flags which display the available flags for compilation.
Compiler options –xhelp=flags and –xhelp=readme

$cc –xhelp=flags
When in doubt, use the –v option to see the macro expansions. Always use –xdryrun option to see the expansions, but not for the compilation.

$cc –c –fast –xdryrun program.c
-xlist options are best to know the syntax errors.

For the better performance always use the –xarch flag

This command should be used to tune a specific processor:

$-xchip

If you want to know the CPU speed then use

$fp version

Information about the processor can be found at:

$psrinfo –v

5.1.3 Floating point exceptions
If a value is divisible by zero, then typically a trap handler will be activated. The –ftrap compiler option should be used to specify which exception you want to trap onto:

Syntax: -ftrap =”t”.
More information can be found in the man pages.

5.1.4 Instruction set architecture
There are two solaris commands to find the information about instruction set architecture.

$isanlist - Displays native instruction sets executable on this platform
$isasinfo - Describes instruction set architectures.

5.1.5 DBX debugger

dbx is one of the debugger provided by the SUN STUDIO compiler.

When running the program, the debugger should be tried.
dbx is one of the best tools to know about any memory leaks in the program.

dbx is invoked at runtime, start typing $ dbx program name. Before running dbx compile the program with -g option. This option includes the source file with debugging code.
$dbx a.out
/* here it will check some library files, then it will be displayed like this
Compilers & tools

(dbx) type check -all
This will invoke RTC (runtime checking)
(dbx) dbx <Arguments>
/* this will read the binary file which contains debugging code, this will display at which line there is a
run time error.*/
(dbx)<Arguments> /*this will display the memory leaks, possible memory leaks.*/

For getting out of the dbx debugger use (dbx) quit.

There is one more debugger which shows the errors in a graphical user interface.

5.1.6 TOTALVIEW debugger

'Total view' debugger is highly efficient in finding out the runtime errors. This is not included with the
sun studio software. It will be invoked with this command.

$totalview  . /program <Arguments>

This will invoke a graphical user interface. If there is a runtime error in the code it will display the line
number.

We can also know how the memory's utilization and CPU utilization are acting at kernel level. There is
a tool called Dtrace.

5.1.7 SUN performance analyzer

To see the results there is er_print command which is command line interface. For GUI use analyzer
command.
$er_print (command line interface)
$analyzer (GUI)

Invoking analyzer can be done directly on UNIX type machines. If trying from windows machine, it
has to do X11 forwarding.

For viewing the results in analyzer, it has to collect the information. This can be done with collect
command.

$collect program
/* this will produce s test.1.er file which is experiment file.
$ analyzer.
Evaluation of parallel programming standards for embedded high performance computing

5.1.8 SUN thread analyzer

Data races can also be viewed by using Sun thread analyzer or Intel thread checker.

For viewing the data races, it has to be compiled with -xinstrument=datarace, but without using -c option.

$collect -r all source <Arguments>
This will produce the .1.er file
To see the data races on GUI,

$tha the .1.er

5.2 MPI compilation

MPI programs can be compiled using MPI libraries.

Compilation:
$mpicc – for c program
$mpiCC – for c++ program

If a program is multithreaded, it’s better to use thread safe MPI.Use –lmpi or –mt for thread safe at compilation time.

$mpicc program.c –o program

When using SUN HPC cluster tools, compile program using
$mpcc --c program
$mpCC --c++ program.

Running MPI program:

$mprun –np n. /program <Arguments>
6. OpenMP & MPI Program demonstration

Before we start testing the benchmarks, we have to think how efficiently the program was written for a single processor. If we get bad results on a single processor then how can we expect good results on several processors? We get bad results due to accessing of memory in a wrong way. This will cause a lot of TLB (Translation look aside buffer) misses and Cache misses. Performance issues depend on different factors. First of all it mainly depends on the user algorithm, then on the operating system, the software libraries and tools. Interconnect is the major bottleneck in Distributed memory machines. But this is not a problem in centralized memory machine as memory is accessed uniformly.

An important task for the parallel programmer is to find out various levels of parallelism. It is up to the programmer to decide at which of the levels it is best to parallelize.

As an introduction how parallel programming is done in OpenMP and MPI, we start with demonstration of simple matrix program.

6.1 OpenMP Program

This program creates two matrices, referred to as A matrix and B matrix respectively and then it will create another matrix copying all the values of A matrix and then adding A and B matrix and copying that to another matrix. Memory is allocated dynamically using mallocs.

In this program we have parallelized the inner loops.

/*Header file inclusion*/
#include <stdio.h>
#include <omp.h>
double omp_get_wtime(void);
void main()
{
  Scanf (" %d", &n);
  double start;
  double end;
  {
    /* allocating memory for the array dynamically */
  }

  nthreads=omp_get_num_procs ();
  printf("No of threads (processors) available on system
  %d\n", nthreads);
  #pragma omp parallel shared (nthreads)
  {
    #pragma omp single
    {

Evaluation of parallel programming standards for embedded high performance computing

```c

Printf("Number of threads used: %d\n", omp_get_num_threads());

start = omp_get_wtime ();
for (h=0; h< 10*n; h++)
{
    #pragma omp parallel for first private (n) shared (a)
    private (j, i)
    For (i=0; i<n; i++)
    {
        For (j=0; j<n; j++)
        {
            a[i][j]=i*10+j;
        }
    }
}

/**** second matrix****/

#pragma omp parallel for first private (n) shared (c) private (k, l)
for(k=0; k<n; k++)
{
    for(l=0; l<n; l++)
    {
        c[k][l]=k+l*100;
    }
}

#pragma omp parallel for first private (n) shared (b) private (p, o)
for(p=0; p<n; p++)
{
    for(o=0; o<n; o++)
    {
        b[p][o]=a[p][o];
    }
}

#pragma omp parallel for shared (d, n, a, c) private (q, r)
for(q=0; q<n; q++)
{
    for(r=0; r<n; r++)
    {
        d[q][r]=a[q][r]+c[q][r];
    }
}
```

1. `nthreads` is a variable and using `omp_get_num_procs()` one will obtain the maximum number of processors that are available on your systems.
2. It will print the number of processors that are available.
3. Here we are using pragma `omp parallel` which tells the compiler to do the work in parallel from here.
4. Here we have used Pragma `omp single`, this tells compiler to do this `printf` statement work on a single processor, which must be the master thread.
5. This will print out how many processors are currently in use for this program.
6. `omp_get_wtime()` is a timing routine used to know the time taken for the program.
7. Here we are creating first matrix. Pragma ‘omp parallel for’ is a parallel work sharing construct. Whenever the compiler sees this `#pragma` it will understand that it has to be divided on threads. Then it will check for the runtime routine or environment variable to know on how many threads to parallelize the data. So there will be a shared value and a private value. So every processor gets the copy of shared value and the private value can be accessed by the particular thread. Here `n` is shared or first private because every time we are updating the value of `n`. So every processor can access this shared value and any processor can update the value. ‘a’ is shared; ‘j’ and ‘I’ are private because it owns the copy of the data. Here we are parallelizing the inner ‘for’ loop which will be having highest communication cost but the load balancing problem will be less. Order of execution will be of any order starting from 0 to N-1.
8. When saying no wait, threads will not wait for other threads; they will get some other work. But there will be a wrong output due to some TLB (Translation look aside buffer) misses. It will cause wrong execution. So it’s better to use barrier at this place so that all threads will wait at this position. Even if you don’t say `#pragma omp barrier` then also it will wait at the place because there is no `nowait` given while parallelizing. At this point all data will be collected. No wait is not allowed in parallel for directive.

Program 1

Serial execution time on Sunfire V220 is 30.98165 sec
After Compiling and executing the above parallel program these are the results we get:

Input:
Please enter the size of array: 100
No of threads (processors) available on system--- 6
Number of threads used: 1
Output: TIME TAKEN IS 35.380995 sec

Repeating this for 2, 3, 4, 5, and 6… threads gives time taken for execution according to table 6.1, illustrated in Figure 6.1
Evaluation of parallel programming standards for embedded high performance computing

The time taken in the case of 2 processors was expected to be, at best half the time taken by a single processor, but when we look at the table the speedup is more than 2. This is because, when a parallel program is running on one thread, it includes the serial execution time and parallel instructions. Parallel instructions include library calls and bookkeeping. So we are giving extra amount of work and some increase in overhead. This results in an increase in parallel execution time on single thread. There is a super linear speedup up to thread 3.

Speedup = $T(p) / p$
Super linear speed up = $S(p) > p$

$T(p)$ = Time taken by ‘p’ processors.
$S(p)$ = Speedup of ‘p’ processors

$p$ = Number of processors.

<table>
<thead>
<tr>
<th>Number of threads used</th>
<th>Time taken(seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35.380995</td>
</tr>
<tr>
<td>2</td>
<td>15.830757</td>
</tr>
<tr>
<td>3</td>
<td>10.877151</td>
</tr>
<tr>
<td>4</td>
<td>8.855770</td>
</tr>
<tr>
<td>5</td>
<td>7.579314</td>
</tr>
<tr>
<td>6</td>
<td>6.986965</td>
</tr>
</tbody>
</table>

Table 6.1 Time taken for execution Figure 6.1 plot Number of processors Versus Time taken

6.2 MPI Program

6.2.1 Parallel programming in MPI

MPI programs are typically run on private memory machines. There will be no shared data. Every processor has its own copy of data. Processors communicate amongst themselves by sending and receiving data through communicator.

The following paragraph is a simple example of how MPI processes will be going on

- This is a story about 10 people.
- They understand the same language and share the same work among each other.
- Their names are 1, the master and 2,3,4,5,6,7,8,9,10 the slaves.
- They work as a team.
- 2 to 10 will listen to everything that 1(master) tells them to do.
- Master decides to distribute the work so that they will be comfortable doing it and the instructions to all the nine workers will not be given separately as they are one group.
OpenMP & MPI Program demonstration

- The program is written in such a way that even if their group increases or decreases the program will not change.

```c
int main (argc, char *argv [])
{
    int rank, size
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
    MPI_Comm_size (MPI_COMM_WORLD, &process Size);

    Iteration = 10 / process Size;

    if (myrank == 1)
    {
        for (int i = 0; i < processSize; i++)
        {
            This was the task
            no = rand(100);
            MPI_SEND(no, 1, int, i, 1, MPI_COMM_WORLD);
        }
    }
    else
    {
        MPI_Recv(no, 1, int, 1, 1, MPI_COMM_WORLD);
        VectorArray = find Factors(no);
        MPI_Send(vectorArray, arraysize, vector, 1, 2,
                  MPI_COMM_WORLD);
    }
    if(rank == 1)
    {
        for (int i=0; i < processSize; i++)
        {
            MPI_Recv(vectorArray, arraysize, vector,
                      MPI_COMM_ANY, 1, MPI_COMM_WORLD);
        }
    }
    MPI_Finalize ();
    Do_work_with_this_vectors ();
    return 0;
}
```

1. The amount of work that this group will perform has to be decided. So they decided to divide the work size by the number of members in their group.
2. 10(Iterations) is just a number, there is no significance to it; It is as good as the amount of work that has to be performed.
3. Now name 1 also known as master has to take responsibility to take up the work and divide it among his friends (you can say it as slaves).
4. Master has a simple strategy to divide the work amongst all the members. He has to call each one by their respective names and assign them their task.
5. Call them one by one and give it to them.
6. Now those whose name is not 1 i.e who is not master, their job was to take up the work.
7. Receive data from process whose rank is 1
8. Consider there is a function which finds the factors of the given number and returns a vector array which stores it.
9. Send this vector array to master, change the tag value so that there will not be any conflict situation.
10. Receive factor of random number from any rank whichever process has finished finding factors.

### 6.2.2 MPI Hello World Program

Here is a simple program to start MPI. This will show the number of processors and their names.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main (int argc, char* argv [])
{
    int myrank, no_of_processors, no_of_row_to_send_to_send,
        row_to_send,extra;
    char * pronoame= (char *)
        malloc(sizeof(char)*MPI_MAX_PROCESSOR_NAME);
    int pronoame_length;
    MPI_Init (&argc, &argv);
    MPI_Comm_world is used to create a MPI network
    MPI_Comm_size (MPI_COMM_WORLD, &no_of_processors);
    MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
    MPI_Get_processor_name (pronoame, &pronoame_length);
    printf("Hello world from processor %d and my name is %s",
        myrank, pronoame);
    return 0;
}
```

$ mpcc hello.c -o hello -lmpi_mt
$ mprun -np 3 ./hello

Hello world from processor 0 and my name is sunc23
Hello world from processor 1 and my name is sunc23
Hello world from processor 2 and my name is sunc23
1. myrank & no_of-processors contain MPI information. myrank contains each individual processor number. no_of_processors will contain total number of processors allocated for the program.
2. prono variable stores processes name.
3. Initialization of MPI process.
4. MPI_Comm_size initializes the no_of_processors that are allocated for the program.
5. MPI_Finalize is used to terminate the MPI processes.
6. MPI compilation.
7. MPI execution.
Evaluation of parallel programming standards for embedded high performance computing
7. Benchmark Test Cases

Here we are going to test some benchmarks which are similar to the kind of processing that is used in radar signal processing where large amount of data is to be stored and accessed fast. The program given for testing is a 4000*4000 matrix, where each block is filled with 640mb of data. Data is read across the rows and is distributed to the processors (threads). The distribution of the work is taken up by the openmp compiler. The OpenMP compiler will understand the #pragma directives. Thus distribution of the work is taken up by a single processor for each distributed work.

In this test program, data is divided and read across different memories of processors. The data in the matrix is read in a tilted way and put the values in a new matrix. The figures below show how it will be done.

We are testing two test cases in the given benchmark program, one is the a_method and the other is the b_method. The a_method uses a 4*4 filter to create one outdata value. The b_method uses a 1*1 filter to create one outdata value. Two buffers are created, one for indata and one for outdata. Which indata should pass on to outdata is controllable. Figure 7.1, Figure 7.2 and Figure 7.3 shows how data is read.

Figure 7.1 Reading data across rows

Figure 7.2 Reading along different memories

Figure 7.3 Process of computation
Here is the sequential C program for the benchmark.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    int buffer_ptr = malloc(rows * sizeof(double *));
    int out_buffer_ptr = malloc(rows * sizeof(double *));
    double table[table_length];
    double *table_zero_ptr = table[(int)((0.5)*(table_length-1)+0.5)];
    double a_table[a_table_steps][a_table_length];

    for (it=0; it<iterations; ++it) {
        for (out_row=0; out_row <rows; ++out_row) {
            row_index_dec = (double) (out_row);
            col_index_dec = 0.0;
            for (out_col =0; out_col < cols;++out_col)
                Switch (method) {
                    Case 1: a_method {
                    
                a_method uses a 4*4 filter to create an out value. For this it uses 16 nearest values along the reading line. The nearest value to the reading line is computed among the four of these invalues and that value is taken in out_buffer.

                Let's take if the row value is 2000, then
                Here we have some initialization of start and stop of loop within the matrix

                for (row_ipol_ind=row_ipol_ind_start; row_ipol_ind_start; ++row_ipol_ind)

                So the value here it computes

                for (row_ipol_ind=1999; 1999<2003, ++row_ipol_ind);

                Case 2: b_method

                The b_method uses only one invalue to create an out value.
                row_index_dec += delta_row;
                col_index_dec += delta_col;

```
In the above program the rows will be distributed among the processors when parallelizing the program. In OpenMP which runs on shared memory machines, the data is divided on to number of processors and shared data can be accessed or modified by any processor in the team.

When it comes to MPI, rows are divided on to the number of processes and these processes are given a rank. We use scatter and gather function to scatter and gather the work. Figure 7.4 shows how data will be scattered on process.

![Figure 7.4 Scattering the work on multiple processes and reading data across different memories.](image)

In the above figures, the first figure shows a row matrix and the second figure a tilted matrix. So we scatter the work on the multiple processes. We are reading data matrices in a tilted way. When tilted, the first tilted data line is read by process 1 and the second will be done by process 2 and so on. So data across the line is read. For getting the exact output we cannot access data directly from other processors memories. Instead if we can get that memory data into memory (main memory) then we can access it. i.e copy it from that location to the main memory location and then use it. This will be done internally by MPI.

For fetching the result in other process memories, it has to execute some more instructions. This will increase the computation time. As for our MPI benchmark program it is doing the same, before creating an output matrix it is fetching data (MPI_Gather) from other process and putting it in output matrix. All this can be taken care of by one process (Master process).

To minimize the communication overhead and to speed up the synchronization we divide the work where it is much suitable. In the given benchmark, shared and private variables are defined to minimize the communication overhead in the program. Once the shared and private variables are defined, the program will communicate through shared variables. MPI and OpenMP are two different programming interfaces and the kinds of communications involved in both of them are different.
Evaluation of parallel programming standards for embedded high performance computing

### 7.1 OpenMP Results

Running on SUNFIRE E2900, ULTRASPARC IV dual core processors running 1.2 GHz contains 12 processors.

1) Test Case 4000 0 0 1 10 (rows, method, delta_row, delta_col, iterations)

$. /benchmark 4000 0 0 1 10
rows:4000
cols:4000
method:0(a_method which uses 4*4)
delta_row:0
delta_col:1
iterations: 10

\[(\Delta\text{row})^2 + (\Delta\text{column})^2 = (0)^2 + (1)^2 = 1\]

**Speedup:** Sequential runtime / Parallel runtime

**Efficiency:** Speedup/ Number of processors.

Sequential runtime=74.74022

In the block shown above, arrows indicate reading direction of the input buffer. The data read along the reading direction will be placed in the output buffer as row by row.

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Time taken in sec</th>
<th>Speed up</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75.590</td>
<td>0.988</td>
<td>0.988</td>
<td>0.011</td>
</tr>
<tr>
<td>2</td>
<td>37.792</td>
<td>1.925</td>
<td>0.962</td>
<td>0.038</td>
</tr>
<tr>
<td>3</td>
<td>29.865</td>
<td>2.436</td>
<td>0.812</td>
<td>0.231</td>
</tr>
<tr>
<td>4</td>
<td>23.341</td>
<td>3.117</td>
<td>0.779</td>
<td>0.283</td>
</tr>
<tr>
<td>5</td>
<td>15.762</td>
<td>4.615</td>
<td>0.923</td>
<td>0.083</td>
</tr>
<tr>
<td>6</td>
<td>15.956</td>
<td>4.559</td>
<td>0.759</td>
<td>0.315</td>
</tr>
<tr>
<td>7</td>
<td>15.658</td>
<td>4.646</td>
<td>0.663</td>
<td>0.506</td>
</tr>
<tr>
<td>8</td>
<td>15.399</td>
<td>4.724</td>
<td>0.590</td>
<td>0.693</td>
</tr>
<tr>
<td>9</td>
<td>15.242</td>
<td>4.773</td>
<td>0.530</td>
<td>0.885</td>
</tr>
<tr>
<td>10</td>
<td>8.699</td>
<td>8.363</td>
<td>0.836</td>
<td>0.195</td>
</tr>
<tr>
<td>11</td>
<td>8.611</td>
<td>8.449</td>
<td>0.768</td>
<td>0.301</td>
</tr>
<tr>
<td>12</td>
<td>8.582</td>
<td>8.477</td>
<td>0.706</td>
<td>0.415</td>
</tr>
</tbody>
</table>
If we take a look at the results, there is a sudden increase in the communication overhead after 5 processors and the efficiency went on decreasing. Increasing the number of threads makes increase in communication overhead, because threads need to communicate with each other by sending data. If threads need to communicate inside the parallel region, then there will be more increase in parallel overhead.

The benchmark is parallelized at highest level where the communication cost is less upto 5 processors. At low level, communication cost is very high which increases the communication overhead but the load balancing problem is less when compared to highest level. Upto 5 processors, the graph above shows almost a linear speedup but decreases after that. This is because of load balancing problem. Load balancing problem arises because the amount of work that the threads will get is not the same. So threads need different amounts of time to finish up their work. But in parallel program, all threads have to wait at some level. This makes increase in overhead and the program does not scale well.

So increasing the number of threads makes increase in communication overhead and load balancing problem. This makes the program inefficient and does not scale well after several processors.
Evaluation of parallel programming standards for embedded high performance computing

2) Test Case 4000 0 0.2588 0.9659 10

\[(\Delta\text{row})^2 + (\Delta\text{column})^2 = (0.2588)^2 + (0.9659)^2 = 1\]
Sequential time = 73.154 sec
Here the matrix will be read in tilted way across different memories.

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Time taken in sec</th>
<th>Speed up</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>73.315</td>
<td>0.997</td>
<td>0.997</td>
<td>0.002</td>
</tr>
<tr>
<td>2</td>
<td>37.820</td>
<td>1.934</td>
<td>0.671</td>
<td>0.033</td>
</tr>
<tr>
<td>3</td>
<td>30.287</td>
<td>2.415</td>
<td>0.805</td>
<td>0.242</td>
</tr>
<tr>
<td>4</td>
<td>23.028</td>
<td>3.176</td>
<td>0.794</td>
<td>0.259</td>
</tr>
<tr>
<td>5</td>
<td>15.849</td>
<td>4.615</td>
<td>0.923</td>
<td>0.083</td>
</tr>
<tr>
<td>6</td>
<td>15.832</td>
<td>4.620</td>
<td>0.770</td>
<td>0.298</td>
</tr>
<tr>
<td>7</td>
<td>15.575</td>
<td>4.696</td>
<td>0.670</td>
<td>0.490</td>
</tr>
<tr>
<td>8</td>
<td>15.278</td>
<td>4.788</td>
<td>0.598</td>
<td>0.670</td>
</tr>
<tr>
<td>9</td>
<td>15.739</td>
<td>4.647</td>
<td>0.516</td>
<td>0.936</td>
</tr>
<tr>
<td>10</td>
<td>8.695</td>
<td>8.412</td>
<td>0.841</td>
<td>0.188</td>
</tr>
<tr>
<td>11</td>
<td>8.683</td>
<td>8.424</td>
<td>0.765</td>
<td>0.305</td>
</tr>
<tr>
<td>12</td>
<td>8.683</td>
<td>8.424</td>
<td>0.702</td>
<td>0.424</td>
</tr>
</tbody>
</table>
3) Test Case 4000 0 0.5 0.8660 10

Sequential time = 72.82 sec

\[(\Delta \text{row})^2 + (\Delta \text{column})^2 = (0.5)^2 + (0.8660)^2 = 1\]

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>73.716</td>
<td>0.987</td>
<td>0.987</td>
<td>0.012</td>
</tr>
<tr>
<td>2</td>
<td>37.751</td>
<td>1.928</td>
<td>0.964</td>
<td>0.036</td>
</tr>
<tr>
<td>3</td>
<td>30.418</td>
<td>2.393</td>
<td>0.797</td>
<td>0.253</td>
</tr>
<tr>
<td>4</td>
<td>22.945</td>
<td>3.173</td>
<td>0.793</td>
<td>0.260</td>
</tr>
<tr>
<td>5</td>
<td>15.972</td>
<td>4.559</td>
<td>0.911</td>
<td>0.096</td>
</tr>
<tr>
<td>6</td>
<td>16.172</td>
<td>4.502</td>
<td>0.750</td>
<td>0.332</td>
</tr>
<tr>
<td>7</td>
<td>15.843</td>
<td>4.596</td>
<td>0.656</td>
<td>0.523</td>
</tr>
<tr>
<td>8</td>
<td>16.260</td>
<td>4.478</td>
<td>0.559</td>
<td>0.786</td>
</tr>
<tr>
<td>9</td>
<td>15.150</td>
<td>4.806</td>
<td>0.534</td>
<td>0.872</td>
</tr>
<tr>
<td>10</td>
<td>8.657</td>
<td>8.410</td>
<td>0.841</td>
<td>0.188</td>
</tr>
<tr>
<td>11</td>
<td>8.543</td>
<td>8.522</td>
<td>0.774</td>
<td>0.290</td>
</tr>
<tr>
<td>12</td>
<td>8.683</td>
<td>8.386</td>
<td>0.698</td>
<td>0.430</td>
</tr>
</tbody>
</table>
4) Test Case 4000 0 0.7071 0.7071 10

Sequential time = 73.206

$(\Delta \text{row})^2 + (\Delta \text{column})^2 = (0.7071)^2 + (0.7071)^2 = 1$

This is the highest tilting in the test cases and the arrows indicate the reading direction of the input buffer.

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>73.229</td>
<td>0.999</td>
<td>0.999</td>
<td>0.0003</td>
</tr>
<tr>
<td>2</td>
<td>37.790</td>
<td>1.937</td>
<td>0.968</td>
<td>0.032</td>
</tr>
<tr>
<td>3</td>
<td>30.288</td>
<td>2.416</td>
<td>0.805</td>
<td>0.241</td>
</tr>
<tr>
<td>4</td>
<td>23.396</td>
<td>3.128</td>
<td>0.782</td>
<td>0.278</td>
</tr>
<tr>
<td>5</td>
<td>16.436</td>
<td>4.453</td>
<td>0.890</td>
<td>0.122</td>
</tr>
<tr>
<td>6</td>
<td>16.205</td>
<td>4.517</td>
<td>0.752</td>
<td>0.328</td>
</tr>
<tr>
<td>7</td>
<td>15.669</td>
<td>4.671</td>
<td>0.667</td>
<td>0.498</td>
</tr>
<tr>
<td>8</td>
<td>15.925</td>
<td>4.596</td>
<td>0.574</td>
<td>0.740</td>
</tr>
<tr>
<td>9</td>
<td>15.871</td>
<td>4.612</td>
<td>0.512</td>
<td>0.951</td>
</tr>
<tr>
<td>10</td>
<td>8.629</td>
<td>8.483</td>
<td>0.848</td>
<td>0.178</td>
</tr>
<tr>
<td>11</td>
<td>8.597</td>
<td>8.514</td>
<td>0.774</td>
<td>0.291</td>
</tr>
<tr>
<td>12</td>
<td>8.585</td>
<td>8.532</td>
<td>0.711</td>
<td>0.406</td>
</tr>
</tbody>
</table>
5) Test Case 4000 1 0 1 10

Rows: 4000
cols: 4000
method: 1(b_method which uses 1*1)
delta_row: 0
delta_col: 1
iterations: 10
Sequential time=1.1842 sec

Here the reading pattern is the same as testcase1. But in this case it reads only one invalue to create outvalue. Data is read along arrows.

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.210</td>
<td>0.978</td>
<td>0.978</td>
<td>0.022</td>
</tr>
<tr>
<td>2</td>
<td>0.616</td>
<td>1.920</td>
<td>0.960</td>
<td>0.041</td>
</tr>
<tr>
<td>3</td>
<td>0.493</td>
<td>2.398</td>
<td>0.799</td>
<td>0.250</td>
</tr>
<tr>
<td>4</td>
<td>0.369</td>
<td>3.203</td>
<td>0.800</td>
<td>0.248</td>
</tr>
<tr>
<td>5</td>
<td>0.247</td>
<td>4.779</td>
<td>0.955</td>
<td>0.046</td>
</tr>
<tr>
<td>6</td>
<td>0.260</td>
<td>4.539</td>
<td>0.756</td>
<td>0.321</td>
</tr>
<tr>
<td>7</td>
<td>0.248</td>
<td>4.769</td>
<td>0.681</td>
<td>0.467</td>
</tr>
<tr>
<td>8</td>
<td>0.256</td>
<td>4.609</td>
<td>0.576</td>
<td>0.735</td>
</tr>
<tr>
<td>9</td>
<td>0.258</td>
<td>4.576</td>
<td>0.508</td>
<td>0.966</td>
</tr>
<tr>
<td>10</td>
<td>0.146</td>
<td>8.085</td>
<td>0.808</td>
<td>0.236</td>
</tr>
<tr>
<td>11</td>
<td>0.155</td>
<td>7.608</td>
<td>0.691</td>
<td>0.445</td>
</tr>
<tr>
<td>12</td>
<td>0.161</td>
<td>7.347</td>
<td>0.612</td>
<td>0.633</td>
</tr>
</tbody>
</table>
The time taken is less than the previous test cases. This is due to the filters. In the previous test case, we used a method that has to read four invalues to create an outvalue, so it has to filter all the four values before creating an outvalue. But in this case, it has to read only one invalue to create outvalue. As seen in the results, for 1 thread it has taken 1.21 seconds. For 5 threads it has taken 0.24 sec. There is a good speedup up to 5 threads. There is an increase in communication overhead compared to the previous test cases because of less amount of work inside the parallel region. This makes imbalanced workload at thread level. If the amount of work is less, dividing this on more number of processors makes increase in parallel overhead.
### Benchmark Test Cases

6) Test Case 4000 1 0.2588 0.9659 10  
Sequential time = 1.2015

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.210</td>
<td>0.992</td>
<td>0.992</td>
<td>0.007</td>
</tr>
<tr>
<td>2</td>
<td>0.616</td>
<td>1.949</td>
<td>0.974</td>
<td>0.026</td>
</tr>
<tr>
<td>3</td>
<td>0.493</td>
<td>2.435</td>
<td>0.811</td>
<td>0.232</td>
</tr>
<tr>
<td>4</td>
<td>0.371</td>
<td>3.238</td>
<td>0.809</td>
<td>0.235</td>
</tr>
<tr>
<td>5</td>
<td>0.259</td>
<td>4.629</td>
<td>0.925</td>
<td>0.080</td>
</tr>
<tr>
<td>6</td>
<td>0.248</td>
<td>4.842</td>
<td>0.807</td>
<td>0.238</td>
</tr>
<tr>
<td>7</td>
<td>0.259</td>
<td>4.629</td>
<td>0.661</td>
<td>0.512</td>
</tr>
<tr>
<td>8</td>
<td>0.246</td>
<td>4.869</td>
<td>0.608</td>
<td>0.643</td>
</tr>
<tr>
<td>9</td>
<td>0.248</td>
<td>4.838</td>
<td>0.537</td>
<td>0.860</td>
</tr>
<tr>
<td>10</td>
<td>0.149</td>
<td>8.043</td>
<td>0.804</td>
<td>0.243</td>
</tr>
<tr>
<td>11</td>
<td>0.181</td>
<td>6.632</td>
<td>0.602</td>
<td>0.658</td>
</tr>
<tr>
<td>12</td>
<td>0.181</td>
<td>6.611</td>
<td>0.550</td>
<td>0.814</td>
</tr>
</tbody>
</table>

![Graphs showing performance metrics for different numbers of threads.](image)
Evaluation of parallel programming standards for embedded high performance computing

Communication overhead for 9 threads is smaller than in testcase 5, this is because the amount of work that we are parallelizing in testcase 6 is comparatively smaller than in testcase 5. If the problem we are trying to solve is less processors will be in an idle state as they won’t have any work load, thereby there will be decrease in efficiency and increase in communication overhead.

For example, we have a single processor which contains 2MB of data set. The problem we are trying to solve has to fetch 10MB into the data set so that there will be a fast accessing of memory and good speedup. But this is not possible because the data set can fetch only 2MB and remaining should be accessed from the main memory. One solution for this is dividing the work on 5 processors, so that every processor data set can fetch 2 MB of work. If we increase the number of processors to more than 5 then there is wastage in other processors data set and this may cause imperfect load balancing. This will increase the communication overhead and degrade the performance.
7) Test Case 4000 1 0.5 0.8660 10

Sequential time = 1.2065

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.210</td>
<td>0.996</td>
<td>0.996</td>
<td>0.003</td>
</tr>
<tr>
<td>2</td>
<td>0.616</td>
<td>1.957</td>
<td>0.978</td>
<td>0.021</td>
</tr>
<tr>
<td>3</td>
<td>0.491</td>
<td>2.455</td>
<td>0.818</td>
<td>0.221</td>
</tr>
<tr>
<td>4</td>
<td>0.369</td>
<td>3.265</td>
<td>0.816</td>
<td>0.224</td>
</tr>
<tr>
<td>5</td>
<td>0.257</td>
<td>4.677</td>
<td>0.935</td>
<td>0.068</td>
</tr>
<tr>
<td>6</td>
<td>0.257</td>
<td>4.679</td>
<td>0.779</td>
<td>0.282</td>
</tr>
<tr>
<td>7</td>
<td>0.263</td>
<td>4.575</td>
<td>0.653</td>
<td>0.529</td>
</tr>
<tr>
<td>8</td>
<td>0.246</td>
<td>4.885</td>
<td>0.610</td>
<td>0.637</td>
</tr>
<tr>
<td>9</td>
<td>0.246</td>
<td>4.900</td>
<td>0.544</td>
<td>0.836</td>
</tr>
<tr>
<td>10</td>
<td>0.156</td>
<td>7.698</td>
<td>0.769</td>
<td>0.298</td>
</tr>
<tr>
<td>11</td>
<td>0.153</td>
<td>7.842</td>
<td>0.712</td>
<td>0.402</td>
</tr>
<tr>
<td>12</td>
<td>0.150</td>
<td>7.995</td>
<td>0.666</td>
<td>0.500</td>
</tr>
</tbody>
</table>
8) Test Case 4000 1 0.7071 0.7071 10
Sequential time = 1.1921

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.21029</td>
<td>0.984</td>
<td>0.984</td>
<td>0.015</td>
</tr>
<tr>
<td>2</td>
<td>0.614297</td>
<td>1.940</td>
<td>0.970</td>
<td>0.030</td>
</tr>
<tr>
<td>3</td>
<td>0.49325</td>
<td>2.416</td>
<td>0.805</td>
<td>0.241</td>
</tr>
<tr>
<td>4</td>
<td>0.371258</td>
<td>3.210</td>
<td>0.802</td>
<td>0.245</td>
</tr>
<tr>
<td>5</td>
<td>0.248461</td>
<td>4.797</td>
<td>0.959</td>
<td>0.042</td>
</tr>
<tr>
<td>6</td>
<td>0.260722</td>
<td>4.572</td>
<td>0.762</td>
<td>0.312</td>
</tr>
<tr>
<td>7</td>
<td>0.259871</td>
<td>4.587</td>
<td>0.655</td>
<td>0.525</td>
</tr>
<tr>
<td>8</td>
<td>0.248154</td>
<td>4.803</td>
<td>0.600</td>
<td>0.665</td>
</tr>
<tr>
<td>9</td>
<td>0.253575</td>
<td>4.701</td>
<td>0.522</td>
<td>0.914</td>
</tr>
<tr>
<td>10</td>
<td>0.155874</td>
<td>7.647</td>
<td>0.764</td>
<td>0.307</td>
</tr>
<tr>
<td>11</td>
<td>0.15191</td>
<td>7.847</td>
<td>0.713</td>
<td>0.401</td>
</tr>
<tr>
<td>12</td>
<td>0.158</td>
<td>7.544</td>
<td>0.628</td>
<td>0.590</td>
</tr>
</tbody>
</table>
7.2 MPI Results
Running on E2900 that contains 8 nodes of 12 ULTRASPARC IV dual core processors running at 1.2 GHz using SUN HPC cluster tools 5

1) Test Case 4000 0 0 1 10(using a_method, 4*4 filter)
Sequential time = 197.523

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>100.888</td>
<td>1.957</td>
<td>0.978</td>
<td>0.021</td>
</tr>
<tr>
<td>3</td>
<td>67.710</td>
<td>2.917</td>
<td>0.972</td>
<td>0.028</td>
</tr>
<tr>
<td>4</td>
<td>50.592</td>
<td>3.904</td>
<td>0.976</td>
<td>0.024</td>
</tr>
<tr>
<td>5</td>
<td>40.856</td>
<td>4.834</td>
<td>0.966</td>
<td>0.034</td>
</tr>
<tr>
<td>6</td>
<td>33.812</td>
<td>5.841</td>
<td>0.973</td>
<td>0.027</td>
</tr>
<tr>
<td>7</td>
<td>29.297</td>
<td>6.741</td>
<td>0.963</td>
<td>0.038</td>
</tr>
<tr>
<td>8</td>
<td>25.331</td>
<td>7.797</td>
<td>0.974</td>
<td>0.025</td>
</tr>
<tr>
<td>9</td>
<td>22.969</td>
<td>8.599</td>
<td>0.955</td>
<td>0.046</td>
</tr>
</tbody>
</table>
When compared with the OpenMP testcase 1(a_method), MPI shows pretty good speedup when the number of processors is increased. It is showing almost linear speedup with the increase of number of processors but this is not the case with OpenMP. Communication overhead in MPI is less when compared to OpenMP because every processor has its own memory and communication inside the parallel region is less. Load balancing problem is less in MPI as the work is equally divided among the processors.

2. Test Case 4000 0 0.2588 0.9659 10
Sequential time = 208.526

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>257.5328</td>
<td>0.809</td>
<td>0.404</td>
<td>1.470</td>
</tr>
<tr>
<td>3</td>
<td>148.8386</td>
<td>1.401</td>
<td>0.467</td>
<td>1.141</td>
</tr>
<tr>
<td>4</td>
<td>92.60246</td>
<td>2.251</td>
<td>0.562</td>
<td>0.776</td>
</tr>
<tr>
<td>5</td>
<td>64.69318</td>
<td>3.223</td>
<td>0.644</td>
<td>0.551</td>
</tr>
<tr>
<td>6</td>
<td>45.73926</td>
<td>4.559</td>
<td>0.759</td>
<td>0.316</td>
</tr>
<tr>
<td>7</td>
<td>34.79038</td>
<td>5.993</td>
<td>0.856</td>
<td>0.167</td>
</tr>
<tr>
<td>8</td>
<td>28.02968</td>
<td>7.439</td>
<td>0.929</td>
<td>0.075</td>
</tr>
<tr>
<td>9</td>
<td>23.22629</td>
<td>8.978</td>
<td>0.997</td>
<td>0.002</td>
</tr>
</tbody>
</table>
When comparing with previous test case (MPI test case 1), we find that the amount of work that has to be parallelized in the program is lower in test case 1 than in MPI test case 2. This results in higher communication overhead in case 1. In MPI test case 2, communication overhead is gradually reduced when parallelized on more processors.

**MPI testcase 1 vs MPI testcase 2**

In MPI testcase 1, efficiency is gradually decreased as the number of processors is increased. This results in increase in communication overhead in program execution. MPI testcase 1 does not contain any complex computations between the parallel regions. This testcase only copies the data from different processes and places it in the new matrix. So the problem of complexity is less when compared with MPI testcase 2.

In MPI testcase 2, there was more communication overhead at the beginning and it was gradually reduced when parallelized on more number of processors. Here the problem we are trying to solve is much more complex than in the MPI testcase 1. The data is read in a tilted way, so data that remains in the other processors has to be copied first in order to gather the information. So copying from different processors will result in execution of a larger number of instructions. As said earlier, one way of solving this problem is to divide the work on more processors so that all the processors can get the same amount of work. This will result in decrease of communication overhead gradually.
Evaluation of parallel programming standards for embedded high performance computing

3. Test Case 4000 0 0.5 0.8660 10  
Sequential time = 113.2909

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>168.504</td>
<td>0.672</td>
<td>0.336</td>
<td>1.974</td>
</tr>
<tr>
<td>3</td>
<td>81.716</td>
<td>1.386</td>
<td>0.462</td>
<td>1.163</td>
</tr>
<tr>
<td>4</td>
<td>49.641</td>
<td>2.282</td>
<td>0.570</td>
<td>0.752</td>
</tr>
<tr>
<td>5</td>
<td>34.409</td>
<td>3.292</td>
<td>0.658</td>
<td>0.518</td>
</tr>
<tr>
<td>6</td>
<td>25.661</td>
<td>4.414</td>
<td>0.735</td>
<td>0.359</td>
</tr>
<tr>
<td>7</td>
<td>20.04</td>
<td>5.653</td>
<td>0.807</td>
<td>0.238</td>
</tr>
<tr>
<td>8</td>
<td>16.178</td>
<td>7.002</td>
<td>0.875</td>
<td>0.142</td>
</tr>
<tr>
<td>9</td>
<td>13.52</td>
<td>8.379</td>
<td>0.931</td>
<td>0.074</td>
</tr>
</tbody>
</table>
Benchmark Test Cases

4. Test Case 4000 0 0.7071 0.7071 10
Sequential time = 87.952

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency in %</th>
<th>Communication Overhead in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>147.717</td>
<td>0.595</td>
<td>0.297</td>
<td>2.359</td>
</tr>
<tr>
<td>3</td>
<td>74.466</td>
<td>1.181</td>
<td>0.393</td>
<td>1.540</td>
</tr>
<tr>
<td>4</td>
<td>45.301</td>
<td>1.941</td>
<td>0.485</td>
<td>1.060</td>
</tr>
<tr>
<td>5</td>
<td>31.272</td>
<td>2.812</td>
<td>0.562</td>
<td>0.777</td>
</tr>
<tr>
<td>6</td>
<td>23.345</td>
<td>3.767</td>
<td>0.627</td>
<td>0.592</td>
</tr>
<tr>
<td>7</td>
<td>17.932</td>
<td>4.904</td>
<td>0.700</td>
<td>0.427</td>
</tr>
<tr>
<td>8</td>
<td>14.431</td>
<td>6.094</td>
<td>0.761</td>
<td>0.312</td>
</tr>
<tr>
<td>9</td>
<td>11.855</td>
<td>7.418</td>
<td>0.824</td>
<td>0.213</td>
</tr>
</tbody>
</table>
Evaluation of parallel programming standards for embedded high performance computing

5. Test Case 4000 1 0 1 10
Sequential time = 534.632

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>294.748</td>
<td>1.813</td>
<td>0.906</td>
<td>0.102</td>
</tr>
<tr>
<td>3</td>
<td>248.548</td>
<td>2.151</td>
<td>0.717</td>
<td>0.394</td>
</tr>
<tr>
<td>4</td>
<td>189.921</td>
<td>2.815</td>
<td>0.703</td>
<td>0.420</td>
</tr>
<tr>
<td>5</td>
<td>153.342</td>
<td>3.486</td>
<td>0.697</td>
<td>0.434</td>
</tr>
<tr>
<td>6</td>
<td>135.003</td>
<td>3.960</td>
<td>0.660</td>
<td>0.515</td>
</tr>
<tr>
<td>7</td>
<td>112.759</td>
<td>4.741</td>
<td>0.677</td>
<td>0.476</td>
</tr>
<tr>
<td>8</td>
<td>98.174</td>
<td>5.445</td>
<td>0.680</td>
<td>0.469</td>
</tr>
<tr>
<td>9</td>
<td>88.261</td>
<td>6.057</td>
<td>0.673</td>
<td>0.485</td>
</tr>
</tbody>
</table>

In this MPI testcase 5, communication cost is much more than the actual work at the process level, so processes are in idle state but they participate in communication. This is the reason for the increase in the parallel execution time than in previous MPI test cases.
6. Test Case 4000 1 0.2588 0.9659 10

Sequential time = 302.4710

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>158.222</td>
<td>1.911</td>
<td>0.955</td>
<td>0.046</td>
</tr>
<tr>
<td>3</td>
<td>107.963</td>
<td>2.801</td>
<td>0.933</td>
<td>0.070</td>
</tr>
<tr>
<td>4</td>
<td>81.901</td>
<td>3.693</td>
<td>0.923</td>
<td>0.083</td>
</tr>
<tr>
<td>5</td>
<td>69.907</td>
<td>4.326</td>
<td>0.865</td>
<td>0.155</td>
</tr>
<tr>
<td>6</td>
<td>55.767</td>
<td>5.424</td>
<td>0.904</td>
<td>0.106</td>
</tr>
<tr>
<td>7</td>
<td>49.101</td>
<td>6.160</td>
<td>0.880</td>
<td>0.136</td>
</tr>
<tr>
<td>8</td>
<td>45.080</td>
<td>6.709</td>
<td>0.838</td>
<td>0.192</td>
</tr>
<tr>
<td>9</td>
<td>41.913</td>
<td>7.216</td>
<td>0.801</td>
<td>0.247</td>
</tr>
</tbody>
</table>

When b_method is compared with previous a_method MPI testcases, the problem we are trying to solve is lessened thereby increasing the parallel overhead gradually. This is because data sets are not used in right way.
Evaluation of parallel programming standards for embedded high performance computing

7. Test case 4000 1 0.5 0.8660 10
Sequential time = 291.761

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency in %</th>
<th>Communication Overhead in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>159.731</td>
<td>1.826</td>
<td>0.913</td>
<td>0.094</td>
</tr>
<tr>
<td>3</td>
<td>109.555</td>
<td>2.663</td>
<td>0.887</td>
<td>0.126</td>
</tr>
<tr>
<td>4</td>
<td>84.280</td>
<td>3.461</td>
<td>0.865</td>
<td>0.155</td>
</tr>
<tr>
<td>5</td>
<td>67.761</td>
<td>4.305</td>
<td>0.861</td>
<td>0.161</td>
</tr>
<tr>
<td>6</td>
<td>57.280</td>
<td>5.093</td>
<td>0.848</td>
<td>0.177</td>
</tr>
<tr>
<td>7</td>
<td>49.647</td>
<td>5.876</td>
<td>0.839</td>
<td>0.191</td>
</tr>
<tr>
<td>8</td>
<td>45.159</td>
<td>6.460</td>
<td>0.807</td>
<td>0.238</td>
</tr>
<tr>
<td>9</td>
<td>41.767</td>
<td>6.985</td>
<td>0.776</td>
<td>0.288</td>
</tr>
</tbody>
</table>
### Benchmark Test Cases

8) Test Case 4000 1 0.7071 0.7071 10
Sequential time = 291.475

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Time taken in sec</th>
<th>Speedup</th>
<th>Efficiency in %</th>
<th>Communication Overhead in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>159.066</td>
<td>1.832</td>
<td>0.916</td>
<td>0.091</td>
</tr>
<tr>
<td>3</td>
<td>109.817</td>
<td>2.654</td>
<td>0.884</td>
<td>0.130</td>
</tr>
<tr>
<td>4</td>
<td>83.630</td>
<td>3.485</td>
<td>0.871</td>
<td>0.147</td>
</tr>
<tr>
<td>5</td>
<td>67.591</td>
<td>4.312</td>
<td>0.862</td>
<td>0.159</td>
</tr>
<tr>
<td>6</td>
<td>57.763</td>
<td>5.046</td>
<td>0.841</td>
<td>0.189</td>
</tr>
<tr>
<td>7</td>
<td>49.517</td>
<td>5.886</td>
<td>0.840</td>
<td>0.189</td>
</tr>
<tr>
<td>8</td>
<td>44.087</td>
<td>6.611</td>
<td>0.826</td>
<td>0.210</td>
</tr>
<tr>
<td>9</td>
<td>41.704</td>
<td>6.989</td>
<td>0.776</td>
<td>0.287</td>
</tr>
</tbody>
</table>

"Test Case 4000 1 0.7071 0.7071 10"
Evaluation of parallel programming standards for embedded high performance computing

7.3 CPU Utilization

Finding exact CPU utilization is difficult in real time. But, there is a way to find out exact CPU utilization when a process is executed.

Top is linux command to know the CPU utilization.

$top

CPU states: 25.3% idle, 68.1% user, 6.6% kernel, 0.0% iowait, 0.0% swap
Memory: 48G real, 43G free, 2449M swap in use, 41G swap free

<table>
<thead>
<tr>
<th>PID</th>
<th>USERNAME</th>
<th>LWP</th>
<th>PRI</th>
<th>NICE</th>
<th>SIZE</th>
<th>RES</th>
<th>STATE</th>
<th>TIME</th>
<th>CPU</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>25414</td>
<td>gp969722</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>276M</td>
<td>252M</td>
<td>cpu/531</td>
<td>0:04</td>
<td>0.83%</td>
<td>a.out (Master)</td>
</tr>
<tr>
<td>25415</td>
<td>gp969722</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>276M</td>
<td>252M</td>
<td>cpu/1</td>
<td>0:04</td>
<td>0.82%</td>
<td>a.out (Slave)</td>
</tr>
<tr>
<td>25418</td>
<td>gp969722</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>276M</td>
<td>252M</td>
<td>cpu/530</td>
<td>0:04</td>
<td>0.81%</td>
<td>a.out (Slave)</td>
</tr>
<tr>
<td>25421</td>
<td>gp969722</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>276M</td>
<td>252M</td>
<td>cpu/16</td>
<td>0:04</td>
<td>0.81%</td>
<td>a.out (Slave)</td>
</tr>
<tr>
<td>25420</td>
<td>gp969722</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>276M</td>
<td>252M</td>
<td>cpu/522</td>
<td>0:04</td>
<td>0.81%</td>
<td>a.out (Slave)</td>
</tr>
<tr>
<td>25419</td>
<td>gp969722</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>276M</td>
<td>252M</td>
<td>cpu/512</td>
<td>0:04</td>
<td>0.81%</td>
<td>a.out (Slave)</td>
</tr>
<tr>
<td>25417</td>
<td>gp969722</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>276M</td>
<td>252M</td>
<td>cpu/17</td>
<td>0:04</td>
<td>0.81%</td>
<td>a.out (Slave)</td>
</tr>
<tr>
<td>25416</td>
<td>gp969722</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>276M</td>
<td>252M</td>
<td>cpu/523</td>
<td>0:04</td>
<td>0.81%</td>
<td>a.out (Slave)</td>
</tr>
</tbody>
</table>

We got the above results while executing the MPI test case 4(a_method). CPU usage will not remain constant throughout the program execution. It will rapidly change during the process execution. The above results show different CPU usages of different processes.

When we look at the CPU utilization of MPI testcase 4(a_method), process ID 25414 is using 0.83% of the CPU capacity while the remaining processes are using 0.81%. This states that PID 25414 is a master process which is doing more work throughout the program execution. It is impossible to give the exact measure of CPU utilization in the program.
8. Conclusions

Based on the comparisons and results, performance depends on various factors in shared memory and private memory systems. In our benchmark program the two filters, a_method (4*4) and b_method (1*1) behave differently. The speedup and communication overhead in the program mainly depend on how efficiently the algorithm has been written and on the type of calculation used in the program. They also depend on the hardware and communication used. Ideally, the speedup should increase linearly with the number of processors but in reality it does not behave linearly as communication overhead, latency and many other problems make parallelization inefficient.

Speedup, based on data parallelism, is achieved by distributing the amount of work evenly on a number of processors. When communication overhead in OpenMP is taken into account, performance depends on accessing of memory and level of parallelization. In the given Benchmark program, on parallelizing at highest level, it is showing good speedup up to a few processors only, since communication cost is less at this level. If the number of processors is increased then there will be an increase in communication overhead and load balancing problems.

When comparing OpenMP with MPI, the load balancing problem is much less in OpenMP. In OpenMP the amount of work to be parallelized is less, which results in more communication overhead when parallelizing the work on more processors. When MPI benchmark is considered, it shows pretty good speedup and low communication overhead with increase in the number of processors.

Measuring the CPU utilization is difficult when the program is being executed. While executing the program the CPU usage changes rapidly depending upon the work load of the processors. One can measure CPU utilization by creating another process of CPU program in main program execution. When a process is created it will invalidate the other processes, until the created process is finished. Thus, we can find CPU utilization. When a work load is smaller, CPU utilization will be less.

There are various factors that affect the performance of shared and private memory systems. One factor to be discussed is Communication latency which mainly depends on the interconnection network. Minimizing communication latency is always the goal. Another factor is throughput at lower bandwidths. Throughput plays major role in providing faster connection.

When it comes to programming efficiency, OpenMP is relatively better than MPI. It contains less number of instructions, lots of help from the compiler, and it is easy to parallelize the code as well. Even the compiler can help in parallelizing the code (works only for FORTRAN and C) by auto scoping.

Further it is easier to learn OpenMP than MPI but OpenMP cannot determine the loop dependencies in the program. When MPI is considered there is no help from the compiler. It is up to the programmer to decide how to distribute the work on processors. In MPI, the user has to make explicit calls in order to parallelize the code. It is easy to manage a network and also easy to increase the number of nodes in MPI.

When considering OpenMP and MPI, High performance and low communication overhead always depend on the operating system, libraries and compilers used for parallelizing the code. It also depends on the type of architecture used in the hardware for communication.
Future work

Future work in OpenMP should mainly focus on providing flexibility in finding the loop dependencies and on synchronization and communication overheads when parallelization is done at highest level where speedup is good up to a few processors. New techniques have to be implemented to decrease the load balancing issues and communication overhead when increasing the number of processors without increasing the program complexity.
9. References


[9]. http://groups.google.com/group/comp.parallel/browse_thread/thread/8ec82249ad050ac8/5778d8fee8008e54?lnk=st&q=difference+between+openmp+and+Mpi&rnum=1&hl=en#5778d8fee8008e54 difference, March 2009-03-11.


Evaluation of parallel programming standards for embedded high performance computing


Appendix A Source code

This appendix contains the benchmark that is similar to many tasks in radar signal processing. The same benchmark is written in C and C++. C++ benchmark is used for OpenMP and C benchmark is used in MPI programming.

A: C++ RADAR signal benchmark

This C++ benchmark program includes two files. One is buffer.h and benchmark.cpp which was written in OpenMP. The buffer.h file is used to create a large array of buffer that will be needed for the program and the benchmark.cpp code contains the original problem that we are trying to solve.

```c
/* File name: buffer.h */

#ifndef BUFFER_H_HEADER
#define BUFFER_H_HEADER
#include <complex.h>
#include <vector>
#include <stdexcept>
#include <complex>
using namespace std;
typedef std::complex<float> vector_complex;
template <class T>
class buffer
{
public:
inline T* operator[](unsigned int row);
buffer(unsigned int rows, unsigned int cols):nof_rows(rows), nof_cols(cols)
{
create_array();
}
virtual ~buffer();

private:
void delete_array();
void create_array();
buffer(const buffer<T> &);
unsigned int nof_rows;
unsigned int nof_cols;
T** array_ptr;
};
template <class T>
T* buffer<T>::operator[](unsigned int row)
```
{  
if (row >= nof_rows)  
{  
throw std::domain_error("buffer:Index outside vector");  
}  
return array_ptr[row];  
}

template <class T>  
buffer<T>::~buffer()  
{  
delete_array();  
}  
template <class T>  
void buffer<T>::create_array()  
{  
double t1 =omp_get_wtime();  
array_ptr=new T*[nof_rows*nof_cols];  
#pragma omp parallel for  
for (int i = 0; i < nof_rows; i++)  
{  
#ifdef PLACEMENT_NEW  
array_ptr[i]=(T*) malloc(nof_cols * sizeof(T));  
T * ptr =array_ptr[i];  
for(int j=0;j<nof_cols;j++,ptr++)  
new(ptr) T();  
#else  
array_ptr[i] =new T[nof_cols];  
#endif  
}  
}  
double t2=omp_get_wtime();  
std::cout " Allocation time was: " << t2-t1 << std::endl;  
}  

template <class T>  
void buffer<T>::delete_array()  
{  
for(int i =0;i< nof_rows;i++)  
{  
#ifdef PLACEMENT_NEW  
T * ptr=array_ptr[i];  
for (int j=0; j < nof_cols;j++,ptr++)  
ptr->~T();  
#else  
delete[] array_ptr[i];  
#endif  
}  
}  

A: 2 C++ Benchmark.cpp

This source code includes OpenMP which are used to parallelize the code. We have two methods which use 4*4 filter and 1*1 filter. The a_method is using 4*4 and b_method is using 1*1 filter.

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <time.h>
#include <math.h>
#include <iomanip>
#include "buffer.h"
#include <omp.h>
using namespace std;

typedef enum {
a_method,
b_method
} ipol_type;
const unsigned int a_table_steps = 1000;
const unsigned int table_length =1001;
const int a_length =4;
const std::complex<float>complex_zero = std::complex<float>(0.0,0.0);

int main(int argc, char* argv[]) {
    double timer, timer1;
    if (argc != 6) {
        std::cout << "wrong no of arguments: should be rows, method, delta_row, delta_col, iterations" << std::endl;
        exit(-1);
    }
    std::cout << "wrong no of arguments: should be rows, method, delta_row, delta_col, iterations" << std::endl;
    exit(-1);
    
    unsigned int rows = std::atoi(argv[1]);
    unsigned int cols = rows;
    ipol_type method = ipol_type(std::atoi(argv[2]));
    float delta_row = std::atof(argv[3]);
    float delta_col = std::atof(argv[4]);
```
unsigned int iterations = std::atoi(argv[5]);

std::cout << std::setw(35) << "rows:" << std::setw(15) << rows << std::endl
<< std::setw(35) << "cols:" << std::setw(15) << cols << std::endl
<< std::setw(35) << "method:" << std::setw(15) << method << std::endl
<< std::setw(35) << "delta_row:" << std::setw(15) << delta_row << std::endl
<< std::setw(35) << "delta_col:" << std::setw(15) << delta_col << std::endl
<< std::setw(35) << "iterations:" << std::setw(15) << iterations << std::endl;

std::cout << "start allocating: " << rows*cols*4*2 + rows*cols*4*2*4 << "Bytes" << std::endl;

int nthreads;
/* printing how many number of threads that are participating in the team */
#pragma omp parallel shared(nthreads)
{
    /* #pragma omp single makes the function std::cout to be run on single processor*/
    #pragma omp single
    std::cout<<"Number of threads in the team:"<<omp_get_num_threads()<<std::endl;
}

buffer<vector_complex>* in_buffer_ptr = new buffer<vector_complex>(2*rows,2*cols);
buffer<vector_complex>* out_buffer_ptr = new buffer<vector_complex>(rows,cols);
std::complex<float> table[table_length];
std::complex<float>* table_zero_ptr = &table[int((0.5)*(table_length-1)+0.5)];
float a_table[a_table_steps] [a_length];
timer = omp_get_wtime();
int out_row, it, out_col, chunk;
float row_index_dec, col_index_dec;

int col_ipol_ind, row_ipol_ind, row_index, col_index, col_index_error, row_index_round, col_index_round;
int num_threads;
num_threads=omp_get_num_threads();

#pragma omp parallel for shared(iterations, rows, cols, num_threads, row_index_dec, delta_row, delta_col, col_index_dec, method, a_table_steps, complex_zero, in_buffer_ptr, out_buffer_ptr, table_zero_ptr, table_length)
private(it, col_index_error, row_index_round, col_index_round, row_index, row_ipol_ind, col_ipol_ind, out_col, out_row)
{
    for (int it=0; it<iterations; ++it)
    {
        
    }
std::cout << "Iteration: " << it << std::endl;

#pragma omp parallel for
shared(iterations, rows, cols, num_threads, row_index_dec, delta_row, delta_col, col_index_dec, method, a_table_steps, complex_zero, in_buffer_ptr, out_buffer_ptr, table_zero_ptr, table_length)
private(it, col_index_error, row_index_round, col_index_round, row_index, row_ipol_ind, col_ipol_ind, out_col, out_row)
for (int out_row = 0; out_row < rows; ++out_row)
{
    float row_index_dec = float(out_row);
    float col_index_dec = 0.0;

    for (int out_col = 0; out_col < cols; ++out_col)
    {
        if (method == a_method)
        {
            int row_index = int(floor(row_index_dec));
            int col_index = int(floor(col_index_dec));
            int row_ipol_ind_start = std::max(row_index - (a_length/2 - 1), 0);
            int col_ipol_ind_start = std::max(col_index - (a_length/2 - 1), 0);
            int row_ipol_ind_stop = std::min(row_index + (a_length/2 + 1), int(rows));
            int col_ipol_ind_stop = std::min(col_index + (a_length/2 + 1), int(cols));
            std::complex<float> acc;
            float *row_a_ptr = &(a_table[int((row_index_dec - row_index)*(a_table_steps - 1) + 0.5)][row_ipol_ind_start - (row_index -(a_length/2 - 1))]);
            for (int row_ipol_ind = row_ipol_ind_start; row_ipol_ind < row_ipol_ind_stop; ++row_ipol_ind)
            {
                acc = complex_zero;
                float *col_a_ptr = &(a_table[int((col_index_dec - col_index)*(a_table_steps - 1) + 0.5)][col_ipol_ind_start - (col_index -(a_length/2 - 1))]);
                std::complex<float>* buf_ptr = &(*in_buffer_ptr)[row_ipol_ind][col_ipol_ind_start];
                for (col_ipol_ind = col_ipol_ind_start; col_ipol_ind < col_ipol_ind_stop; ++col_ipol_ind)
                {
                    acc += *col_a_ptr * *buf_ptr;
                    col_a_ptr++;
                    buf_ptr++;
                }
                (*out_buffer_ptr)[out_row][out_col] += *row_a_ptr*acc;
                row_a_ptr++;
            }
        }
    }
    if (method == b_method)
    {
    }
}
Evaluation of parallel programming standards for embedded high performance computing

```c
int col_index_round = int(col_index_dec + 0.5);
int row_index_round = int(row_index_dec + 0.5);
float col_index_error = col_index_dec - float(col_index_round);
(*out_buffer_ptr)[out_row][out_col] += (*in_buffer_ptr)[row_index_round][col_index_round] * table_zero_ptr[int((col_index_error) * (table_length - 1))];
row_index_dec += delta_row;
col_index_dec += delta_col;
}
}

timer1 = omp_get_wtime();
std::cout << "The Time Taken is: " << timer1 - timer << std::endl;
delete in_buffer_ptr;
delete out_buffer_ptr;
}

A: 3 MPI Benchmark

/*********************************************************
/* File name: MPIbenchmark.c                                */
/**********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <mpi.h>

const unsigned int a_table_steps = 1000;
const unsigned int table_length = 1001;
const int a_length = 4;

int max(int val1, int val2);
int min(int val1, int val2);

int main(int argc, char *argv[])
{
    if (argc != 6)
    {
        printf("Wrong nof arguments: Should be rows, method, delta_row, delta_col, iterations \n");
        exit(-1);
    }

    int myrank, no_of_processors, no_of_row_to_send_to_send, row_to_send, extra;
```
Appendix A Source code

```c
char * proname= (char *) malloc(sizeof(MPI_MAX_PROCESSOR_NAME));
int pronamelength;

MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD,&no_of_processors);

/* Find out my identity in the default communicator */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Get_processor_name(proname, &pronamelength);

double **in_buffer_ptr,**out_buffer_ptr;
signed int m,i,j,rows,cols;
double delta_row;
double delta_col;
signed int iterations;

double process_time,my_machine_time,total_computation_time;

process_time = MPI_Wtime();

rows = atoi(argv[1]);
cols = rows;
int method = atoi(argv[2]);
delta_row = atof(argv[3]);
delta_col = atof(argv[4]);
iterations = atoi(argv[5]);

/*
** Calculating the size of arrays (matrices) to be sent...
** on the basis of the no_of_processors available
*/

no_of_row_to_send_to_send = rows / no_of_processors;

// declaring an extra element, if the rows cannot be equally divided into the processors
extra = rows % no_of_processors;

//Assigning the new size of rows to be send (based on the extra rows, if any)

if(extra > 0 )
    row_to_send = no_of_row_to_send_to_send + 1;
else
```

73
Evaluation of parallel programming standards for embedded high performance computing

\[ \text{row\_to\_send} = \text{no\_of\_row\_to\_send\_to\_send}; \]

if (myrank == 0)
{
    printf("\text{rows} = \%d", rows);
    printf("\text{ncols} = \%d", cols);
    printf("\text{method} = \%d", method);
    printf("\text{ndelta\_row} = \%lf", delta_row);
    printf("\text{ndelta\_col} = \%lf", delta_col);
    printf("\text{niterations} = \%d", iterations);
    printf("\text{nRows to distrubute} = \%d", row\_to\_send);
}

/* allocating memory for the array dynamically */
in_buffer_ptr = malloc (rows * sizeof (double *));
out_buffer_ptr = malloc (rows * sizeof (double *));

for (m = 0; m < rows; m++)
{
    in_buffer_ptr[m] = malloc(rows * sizeof(double));
    out_buffer_ptr[m] = malloc(rows * sizeof(double));
}

for (i = 0; i < rows; i++)
{
    for (j = 0; j < cols; j++)
    {
        in_buffer_ptr[i][j] = i + j * 100;
        //printf(" %lf ", in_buffer_ptr[i][j]);
        out_buffer_ptr[i][j] = i + j * 100;
    }
}

double table[table\_length];
double *table\_zero\_ptr = &table[(int)((0.5)*(table\_length-1)+0.5)];
double a_table[a\_table\_steps][a\_length];
int out\_row, it, out\_col;
double row\_index\_dec, col\_index\_dec, col\_index\_error;
int col\_index\_round, row\_index\_round;
double king, queen;
int row\_index, col\_index, row\_ipol\_ind\_start, col\_ipol\_ind\_start, row\_ipol\_ind\_stop, col\_ipol\_ind\_stop, row\_ipol\_ind, col\_ipol\_ind;
double acc, *row\_a\_ptr, *col\_a\_ptr, *buf\_ptr;

total\_computation\_time = MPI\_Wtime();
for(it=0;it<iterations;++it)
{
/*
 ** Here we actually Scatter the data of matrix_1 to other machine/processors
 ** First we count the number of row_to_send to scatter
 ** that is the number of elements to be taken up by each processor.
*/
MPI_Barrier(MPI_COMM_WORLD);

MPI_Scatter(**in_buffer_ptr,row_to_send,MPI_DOUBLE,**in_buffer_ptr,row_to_send,MPI_DOUBLE,0,MPI_COMM_WORLD);

MPI_Scatter(**out_buffer_ptr,row_to_send,MPI_DOUBLE,**out_buffer_ptr,row_to_send,MPI_DOUBLE,1,MPI_COMM_WORLD);

my_machine_time = MPI_Wtime();
printf("\n Iteration %d by Process :id >> %d >>>>>%s\n",it,myrank,priname);

for (out_row=0;out_row <row_to_send;++out_row)
{
   row_index_dec = (double)(out_row);
   col_index_dec = 0.0;
   for (out_col =0;out_col < cols;++out_col)
   {
      switch (method)
      {
      case 1 : // a_method
      {
         row_index = (int) (floor (row_index_dec));
         col_index = (int) (floor (col_index_dec));

         row_ipol_ind_start = max (row_index - (a_length/2-1), 0);
         col_ipol_ind_start = max (col_index - (a_length/2-1), 0);

         row_ipol_ind_stop = min (row_index + (a_length/2+1), (int) (row_to_send));
         col_ipol_ind_stop = min (col_index + (a_length/2+1), (int) (cols));

         row_a_ptr = &(a_table[(int)((row_index_dec-row_index)*(a_table_steps-1)+0.5)][row_ipol_ind_start-(row_index - (a_length/2-1))]);

         for(row_ipol_ind = row_ipol_ind_start; row_ipol_ind<row_ipol_ind_stop;++row_ipol_ind
         )
         {
         
         }
      }
Evaluation of parallel programming standards for embedded high performance computing

```c
acc = 0.0;
col_a_ptr = &(a_table[(int)((col_index_dec-col_index)*(a_table_steps-1) +0.5)]
[[col_ipol_ind_start-(col_index - (a_length/2-1))]);
buf_ptr = &in_buffer_ptr[row_ipol_ind][col_ipol_ind_start];
for(col_ipol_ind = col_ipol_ind_start; col_ipol_ind<col_ipol_ind_stop; ++col_ipol_ind)
  {
    acc += *col_a_ptr * *buf_ptr;
    col_a_ptr++;  
    buf_ptr++;
  } 
out_buffer_ptr[out_row][out_col] += *row_a_ptr*acc;
row_a_ptr++; 
}
break;
case 2 : //b_method
{ 
col_index_round = (int)(col_index_dec+0.5);
row_index_round = (int)(row_index_dec+0.5);
if(row_index_round > out_row)
  row_index_round = out_row;
col_index_error = col_index_dec -(double)(col_index_round);
out_buffer_ptr[out_row][out_col] +=in_buffer_ptr[row_index_round][col_index_round]*table_zero_ptr[ (int)((col_index_error)*(table_length-1))];
  }
break;
default:
  break;
}
row_index_dec +=delta_row;
col_index_dec +=delta_col;
}
}

my_machine_time = MPI_Wtime() - my_machine_time;

MPI_Gather(&(**in_buffer_ptr),row_to_send,MPI_DOUBLE,&(**in_buffer_ptr),row_to_send,MPI_DOUBLE,0,MPI_COMM_WORLD);

MPI_Gather(&(**out_buffer_ptr),row_to_send,MPI_DOUBLE,&(**out_buffer_ptr),row_to_send,MPI_DOUBLE,1,MPI_COMM_WORLD);

free (in_buffer_ptr);
free (out_buffer_ptr);
```
total_computation_time = MPI_Wtime() - total_computation_time;
process_time = MPI_Wtime() - process_time;

MPI_Barrier(MPI_COMM_WORLD);
printf("\n\nComputation At Process %d >>> %s Time !===> %lf\n", myrank, prname, my_machine_time);
if(myrank == 0)
{
    printf("Complete Process Time !===> %lf\n", process_time);
    printf("Total Computation Time !===> %lf\n", total_computation_time);
}
MPI_Finalize();

int min(int val1, int val2)
{
    if(val1 > val2)
        return val2;
    else
        return val1;
}

int max(int val1, int val2)
{
    if(val1 > val2)
        return val1;
    else
        return val2;
}