

Overview of Parallel Programming at CCR

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

Parallel Algorithms, 2009

A Little Motivation

- TANSTAAFL = There Aint (sic) No Such Thing As A Free Lunch
- Why do we parallel process in the first place?
 - **Performance** - either solve a bigger problem, or to reach solution faster (or both)
- Hardware is becoming (actually it has been for a while now) intrinsically parallel
- Parallel programming is **not** easy. How easy is sequential programming? Now add another layer (a sizable one, at that) for parallelism ...

Quote

"TANSTAAFL!"

–**Robert A. Heinlein**, *The Moon is a Harsh Mistress*

The Parallel Zoo

There are many parallel APIs (not exhaustive, I am sure that I will miss somebody's favorite ...):

- **MPI, PVM, CHARM++**, GA, BSP, ZPL, ...
- **OpenMP, pthreads, HPF**, TBB, shmem, Ct, Windows threads, ...
- **Co-array Fortran, UPC, Java/Titanium**, ...

(grouped somewhat in terms of message-passing, (distributed) shared-memory, and PGAS)

Parallelism at Three Levels

Task - macroscopic view in which an algorithm is organized around separate concurrent tasks

Data - structure data in (separately) update-able “chunks”

Instruction - ILP through the “flow” of data in a predictable fashion

Dependencies

Bear in mind that you always want to minimize the **dependencies** between your concurrent tasks and data:

- On the task level, design your tasks to be as independent as possible - this can also be **temporal**, namely take into account any necessity for ordering the tasks and their execution
- Sharing of data will require synchronization and thus introduce data dependencies, if not race conditions or contention

Approaching the Problem

The first step in deciding where to add parallelism to an algorithm or application is usually to analyze from the point of **tasks** and **data**:

Tasks How do I reduce this problem into a set of tasks that can be executed concurrently?

Data How do I take the key data and represent it in such a way that large chunks can be operated on independently (and thus concurrently)?

An Hour of Planning ...

Taking the time to carefully analyze an existing application, or plan a new one, can really pay off later:

Plan for parallel programming by keeping data and task parallel constructs/opportunities in mind

Analyze an existing or new program to discover/confirm the locations of the time-consuming portions

Optimize by implementing a strategy using data or task parallel constructs

Auto-Parallelization

Often called “implicit” parallelism, some compilers (usually commercial ones) have the ability to:

- automatically parallelize simple (outer) loops (thread-level parallelism, or TLP)
- automatically vectorize (usually innermost loops) using ILP

Note that thread level parallelism is only usable on an SMP architecture

Auto-Vectorization

Like auto-parallelization, a feature of high-performance (often commercial) compilers on hardware that supports at least some level of vectorization:

- compilers finds low-level operations that can operate simultaneously on multiple data elements using a single instruction
- Intel/AMD processors with SSE/SSE2/SSE3 usually limited to 2^1 - 2^3 vector length (hardware limitation that true “vector” processors do not share)
- User can exert some control through compiler directives (usually vendor specific) and data organization focused on vector operations

Auto-Parallelism

Most commercial compilers have some sort of feature to automatically “auto-parallelize” sections of code:

- tries to identify loops than can “safely” be executed in parallel (little or no dependencies between loop iterations)
- thread-level parallelism (TLP)
- some gains can be had - little or no programming effort required

APOs at CCR

Platform	Invocation note
Linux IA64	ifort -parallel -par_report2 ...
Linux x86/x86_64	ifort -parallel -par_report2 ... pgf90 -Mconcur ...

Shared Memory Parallelism

Here we consider “explicit” modes of parallel programming on shared memory architectures:

- HPF, High Performance Fortran
- SHMEM, the old Cray shared memory library
- Pthreads, UNIX ANSI standard (available on Windows as well)
- OpenMP, common specification for compiler directives and API

PTHREADS Availability at CCR

Platform	Invocation (example)
Linux IA64	icc/icpc/gcc/g++
Linux x86/x86_64	pgcc/pgCC/icc/icpc/gcc/g++

PTHREADS

Synopsis: A low-overhead shared address space programming environment.

Target: Shared memory (only, of course).

Current Status: Widely available for UNIX as the ANSI/IEEE POSIX 1003.1 standard (1995).

More Info: LLNL has an excellent tutorial online:

<http://www.llnl.gov/computing/tutorials/threads>

OpenMP

Synopsis: designed for multi-platform shared memory parallel programming in C/C++/FORTRAN primarily through the use of compiler directives and environmental variables (library routines also available).

Target: Shared memory systems.

Current Status: Specification 1.0 released in 1997, 2.0 in 2002, 2.5 in 2005, 3.0 in 2008. Widely implemented by commercial compiler vendors, also in some open-source compilers.

More Info: The OpenMP home page:

<http://www.openmp.org>

OpenMP Availability at CCR

Platform	Version	Invocation (example)
Linux IA64	2.5	ifort -openmp -openmp_report2 ...
Linux x86_64	2.5	ifort -openmp -openmp_report2 ... pgf90 -mp ...
GNU (>=4.2)	2.5	gfortran -fopenmp ...

PVM

Parallel Virtual Machine

Synopsis: message passing model for heterogeneous computing resources.

Target: Distributed and shared memory systems. Most implementations use TCP/IP communications.

Current Status: Still active, but declining as MPI gains wider acceptance.

More Info: The PVM home page at ORNL:

<http://www.csm.ornl.gov/pvm>

Distributed Memory Parallel Programming

In this case, mainly message passing, with a few other (compiler directives again) possibilities:

- PVM, Parallel Virtual Machine
- MPI, Message Passing Interface, has become the dominant API for general purpose parallel programming
- HPF, High Performance Fortran
- Cluster OpenMP, a new product from Intel to push OpenMP into a distributed memory regime
- UPC, Unified Parallel C, extensions to ISO 99 C for parallel programming (**new**)
- CAF, Co-Array Fortran, parallel extensions to Fortran 95/2003 (**new**, officially part of Fortran 2008)

PVM Availability at CCR

Platform	Availability	Version
Linux IA64	Yes	3.4 (SGI MPT)
Linux x86_64	No (available upon request)	

Message Passing Interface

Synopsis: Message passing library specification proposed as a standard by committee of vendors, implementors, and users.

Target: Distributed and shared memory systems.

Current Status: Most popular (and most portable) of the message passing APIs.

More Info: ANL's main MPI site:

www-unix.mcs.anl.gov/mpi

a.k.a High Performance Fortran

Synopsis: Designed to be a portable extension to FORTRAN 90 for data parallel applications.

Target: Shared and distributed memory systems.

Current Status: HPF-2 specification released by HPFF in 1997. Not widely used, but still active.

More Info: The HPF Forum Home Page

<http://www.crpc.rice.edu/HPFF>

Platform	Version(+MPI-2)
Linux IA64	1.2+(C++, MPI-I/O)
Linux x86_64	1.2+(C++, MPI-I/O), 2.0(OpenMPI)

Generally we use ANL's MPICH (and its Myrinet port, MPICH-MX) as our preferred MPI implementation, but I am starting to favor OpenMPI for its ease of use.

Platform	Availability	Invocation (version)
Linux IA64	No	
Linux x86_64	Yes	pgmpf (all)

Unified Parallel C (UPC)

Unified Parallel C (UPC) is a project based at Lawrence Berkeley Laboratory to extend C (ISO 99) and provide large-scale parallel computing on both shared and distributed memory hardware:

- Explicit parallel execution (SPMD)
- Shared address space (shared/private data like OpenMP), but exploits data locality
- Primitives for memory management
- BSD license (free download)
- <http://upc.lbl.gov/>, community website at <http://upc.gwu.edu/>

Simple example of some UPC syntax:

```

1  shared int all_hits[THREADS];
2  ...
3  ...
4  for (i=0; i < my_trials; i++) my_hits += hit();
5  all_hits[MYTHREAD] = my_hits;
6  upc_barrier;
7  if (MYTHREAD == 0) {
8      total_hits = 0;
9      for (i=0; i < THREADS; i++) {
10         total_hits += all_hits[i];
11     }
12     pi = 4.0*((double) total_hits)/((double) trials);
13     printf("Pi estimated to %10.7f from %d trials on %d threads.\n",
14         pi, trials, THREADS);
15 }

```

Co-Array Fortran

Co-Array Fortran (CAF) is an extension to Fortran (Fortran 95/2003) to provide data decomposition for parallel programs (somewhat akin to UPC and HPF):

- Original specification by Numrich and Reid, ACM Fortran Forum, **17**, no. 2, pp 1-31 (1998).
- ISO Fortran committee included co-arrays in next revision to Fortran standard (c.f. Numrich and Reid, ACM Fortran Forum, **24**, no 2, pp 4-17 (2005), Fortran 2008).
- Does not require shared memory (more applicable than OpenMP)
- Early compiler work at Rice:

<http://www.hipersoft.rice.edu/caf/index.html>

- Simple examples of CAF usage:

```

1  REAL, DIMENSION(N)[*] :: X,Y
2  X      = Y[PE]  ! get from Y[PE]
3  Y[PE]  = X      ! put into Y[PE]
4  Y[:]  = X      ! broadcast X
5  Y[LIST] = X    ! broadcast X over subset of PE's in array LIST
6  Z(:)  = Y[:]  ! collect all Y
7  S = MINVAL(Y[:]) ! min (reduce) all Y
8  B(1:M)[1:N] = S ! S scalar, promoted to array of shape (1:M,1:N)

```

- UPC and CAF are examples of partitioned global address space (PGAS) language, for which a large push is being made by AHPARC/DARPA as part of the **Petascale** computing initiative (also one based on java called Titanium)

Specifications

All specs and API descriptions (as well as other useful info) can be found on the web at:

<http://www.openmp.org>

The current OpenMP specification is 2.5, and is available for FORTRAN and C/C++:

- Fortran 2.0 API (11/2000)
- C/C++ 2.0 API (02/2002)
- Fortran/C/C++ 2.5 API (05/2005)
- Fortran/C/C++ 3.0 API (05/2008)

Specification 3.0 should get supported by Intel's version 11 compilers, PGI version 8, Sun Studio 11/08, and GNU version 4.4.

Trivial OpenMP Example

The most common (and simplest) application of OpenMP is simply through compiler directives, as in the following trivial example:

```

1 !OMP DO SCHEDULE(DYNAMIC,CHUNK)
2   do i=1,length
3     z(i)=sqrt(x(i))*exp(-y(i))
4   end do
5 !OMP END DO NOWAIT

```

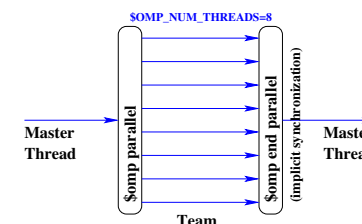
OpenMP 101

- OpenMP goal is to establish & standardize a set of directives for programs on shared memory architectures.
- OpenMP provides programmer with compiler directives to incrementally parallelize an existing serial program (contrast this with the message passing model).
- OpenMP is able to handle both *fine-grain* and *coarse-grain* models of parallelism.

Execution Model

OpenMP introduces multiple execution threads as the work-sharing constructs:

- *Fork-Join*, the master thread spawns a team of threads inside parallel regions.
- *Typical Use*: split compute-intensive loops among the thread team.



OpenMP General Syntax

Most OpenMP constructs are compiler directives or pragmas:

C/C++:

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

F77:

```
C$OMP construct[clause[clause]...]
```

F90:

```
!$OMP construct[clause[clause]...]
```

to compilers that do not support OpenMP, these directives are just **comments**, and have no effect.

OpenMP Construct Categories

OpenMP has a rich feature set:

- Parallel Regions
- Work-sharing
- Synchronization
- Data Environment
- Run-time Library (API)

which I will not discuss further in this talk - we will come back to develop OpenMP in much more detail.

OpenMP Conditional Compilation

- Sometimes you want to use the API, but need a way to hide code when porting to a non-compliant compiler:

```
1 !$   iam = OMP_GET_THREAD_NUM()
2 !$   print*, 'My thread id is ',iam
```

equivalently,

```
1 #ifdef _OPENMP
2     iam = OMP_GET_THREAD_NUM()
3     print*, 'My thread id is ',iam
4 #endif
```

Advantages over Message Passing

OpenMP has several advantages over Message Passing methods:

- Domain decomposition methodology is the same, but implementing it in OpenMP can be easier, as global data can be read without any need for synchronization or message passing.
- Parallelize only parts of the code that require it (profiling is key!). Pre- and post-processing can be left sequential.

Best of Both Worlds?

How about combining OpenMP with Message Passing?

- Message Passing between machines, OpenMP within.
- Allow application dependent mixing within an SMP.
- Coarse grain with Message Passing, fine grain with OpenMP.

Basic Features of Message Passing

Message passing codes run the same (usually serial) code on multiple processors, which communicate with one another via library calls which fall into a few general categories:

- Calls to initialize, manage, and terminate communications
- Calls to communicate between two individual processes (point-to-point)
- Calls to communicate among a group of processes (collective)
- Calls to create custom datatypes

I will briefly cover the first three, and present a few concrete examples.

OpenMP Summary

Brief summary of OpenMP highlights:

- Fine grained parallelism easy to implement, but not very scalable.
- Work-sharing constructs in coarse grained parallel regions for intermediate performance.
- SPMD model offers best scalability (at cost of more work for the programmer) - relatively easy to implement compared to message passing.
- Incremental Parallelization - does not require a complete redesign of the code.

Outline of a Program Using MPI

General outline of any program using MPI:

- 1 Include MPI header files
- 2 Declare variables & Data Structures
- 3 Initialize MPI
- 4 .
- 5 Main program – message passing enabled
- 6 .
- 7 Terminate MPI
- 8 End program

MPI Header Files

All MPI programs need to include the MPI header files to define necessary datatypes.

- In **C**:

```
1 #include "mpi.h"
2 #include <stdio.h>
3 #include <math.h>
```

- In **FORTRAN 77**

```
1 program main
2 implicit none
3 include 'mpif.h'
```

MPI Routines & Their Return Values

Generally the MPI routines return an error code, using the exit status in C, which can be tested with a predefined success value:

```
1 int ierr;
2 ...
3 ierr = MPI_INIT(&argc,&argv);
4 if (ierr != MPI_SUCCESS) {
5     ... exit with an error ...
6 }
7 ...
```

MPI Naming Conventions

MPI functions are designed to be as language independent as possible.

- Routine names all begin with **MPI_**:

- FORTRAN names are typically upper case:

```
call MPI_XXXXXXX(param1, param2, ..., IERR)
```

- C functions use a mixed case:

```
ierr = MPI_Xxxxxxx(param1, param2, ...)
```

- MPI constants are all upper case in both C and FORTRAN:

```
MPI_COMM_WORLD, MPI_REAL, MPI_DOUBLE, ...
```

and in FORTRAN the error code is passed back as the last argument in the MPI subroutine call:

```
1 integer :: ierr
2
3 call MPI_INIT(ierr)
4 if (ierr.ne.MPI_SUCCESS) STOP 'MPI_INIT failed.'
```

MPI Handles

- MPI defines its own data structures, which can be referenced by the use through the use of **handles**.
- handles can be returned by MPI routines, and used as arguments to other MPI routines.
- Some examples:
 - `MPI_SUCCESS` - Used to test MPI error codes. An integer in both C and FORTRAN.
 - `MPI_COMM_WORLD` - A (pre-defined) communicator consisting of all of the processes. An integer FORTRAN, and a `MPI_Comm` object in C.

MPI Datatypes in C

In C, the basic datatypes (and their ISO C equivalents) are:

MPI Datatype	C Type
<code>MPI_FLOAT</code>	float
<code>MPI_DOUBLE</code>	double
<code>MPI_LONG_DOUBLE</code>	long double
<code>MPI_INT</code>	signed int
<code>MPI_LONG</code>	signed long int
<code>MPI_SHORT</code>	signed short int
<code>MPI_UNSIGNED_SHORT</code>	unsigned short int
<code>MPI_UNSIGNED_LONG</code>	unsigned long int
<code>MPI_UNSIGNED</code>	unsigned int
<code>MPI_CHAR</code>	signed char
<code>MPI_UNSIGNED_CHAR</code>	unsigned char
<code>MPI_BYTE</code>	–
<code>MPI_PACKED</code>	–

MPI Datatypes

- MPI defines its own datatypes that correspond to typical datatypes in C and FORTRAN.
- Allows for automatic translation between different representations in a heterogeneous parallel environment.
- You can build your own datatypes from the basic MPI building blocks.
- Actual representation is implementation dependent.
- *Convention*: program variables are usually declared as normal C or FORTRAN types, and then calls to MPI routines use MPI type names as needed.

MPI Datatypes in FORTRAN

In FORTRAN, the basic datatypes (and their FORTRAN equivalents) are:

MPI Datatype	C Type
<code>MPI_INTEGER</code>	integer
<code>MPI_REAL</code>	real
<code>MPI_DOUBLE_PRECISION</code>	double precision
<code>MPI_COMPLEX</code>	complex
<code>MPI_DOUBLE_COMPLEX</code>	double complex
<code>MPI_LOGICAL</code>	logical
<code>MPI_CHARACTER</code>	character*1
<code>MPI_BYTE</code>	–
<code>MPI_PACKED</code>	–

Initializing & Terminating MPI

- The first MPI routine called by any MPI program *must be* **MPI_INIT**, called once and only once per program.

- C:**

```
1 int ierr;
2 ierr = MPI_INIT(&argc,&argv);
3 ...
```

- FORTRAN:**

```
1 integer ierr
2 call MPI_INIT(ierr)
3 ...
```

- A process' rank is used to specify source and destination in message passing calls.
- A process' rank can be different in different communicators.
- `MPI_COMM_WORLD` is a pre-defined communicator encompassing all of the processes. Additional communicators can be defined to define subsets of this group.

MPI Communicators

Definition (MPI Communicator)

A **communicator** is a group of processes that can communicate with each other.

- There can be many communicators
- A given process can be a member of multiple communicators.
- Within a communicator, the **rank** of a process is the number (starting at 0) uniquely identifying it within that communicator.

More on MPI Communicators

Typically a program executes two MPI calls immediately after `MPI_INIT` to determine each process' rank:

- C:**

```
1 int MPI_Comm_rank(MPI_Comm comm, int *rank);
2 int MPI_Comm_size(MPI_Comm comm, int *size);
```

- FORTRAN:**

```
1 MPI_COMM_RANK(comm, rank, ierr)
2 MPI_COMM_SIZE(comm, size, ierr)
```

where `rank` and `size` are integers returned with (obviously) the rank and extent (0:number of processes-1).

Simple MPI Program in C

- We have already covered enough material to write the simplest of MPI programs: here is one in C:

```

1 #include <stdio.h>
2 #include "mpi.h"
3
4 int main( int argc, char **argv)
5 {
6     int ierr,myid, numprocs;
7     MPI_Init(&argc,&argv);
8     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
9     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
10
11     printf("Hello World, I am Process %d of %d\n",
12           myid,numprocs);
13     ierr = MPI_Finalize();
14 }

```

Six Function MPI

Many MPI codes can get away with using only the six most frequently used routines:

- `MPI_INIT` for initialization
- `MPI_COMM_SIZE` size of communicator
- `MPI_COMM_RANK` rank in communicator
- `MPI_SEND` send message
- `MPI_RECEIVE` receive message
- `MPI_FINALIZE` shut down communicator

Simple MPI Program in FORTRAN

- and similarly in FORTRAN:

```

1 program Simp
2
3 include 'mpif.h'
4 integer ierr,myid,numprocs
5
6 call MPI_INIT(ierr)
7 call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ierr)
8 call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
9 write(*,('Hello World, I am Process ',i8,', " of ',i8)')
10 $ myid,numprocs
11 call MPI_FINALIZE(ierr)
12 end program Simp

```

Leveraging Existing Parallel Libraries

One “easy” way to utilize parallel programming resources is through the use of existing parallel libraries. Most common examples (note orientation around standard mathematical routines):

- `BLAS` - Basic Linear Algebra Subroutines
- `LAPACK` - Linear Algebra PACKage (uses BLAS)
- `ScaLAPACK` - distributed memory (MPI) solvers for common LAPACK functions
- `PetSC` - Portable, Extensible Toolkit for Scientific Computation
- `FFTW` - Fastest Fourier Transforms in the West

BLAS

The Basic Linear Algebra Subroutines (BLAS) form a standard set of library functions for vector and matrix operations:

- Level 1 Vector-Vector (e.g. **xdot**, where $x=s,d,c,z$)
- Level 2 Matrix-Vector (e.g. **xaxpy**, where $x=s,d,c,z$)
- Level 3 Matrix-Matrix (e.g. **xgemm**, where $x=s,d,c,z$)

These routines are generally provided by vendors hand-tuned at the level of assembly code for optimum performance on a particular processor. Shared memory versions (multithreaded) and distributed memory versions available.

www.netlib.org/blas

LAPACK

The Linear Algebra PACKage (LAPACK) is a library that lies on top of the BLAS (for optimum performance and parallelism) and provides:

- solvers for systems of simultaneous linear equations
- least-squares solutions
- eigenvalue problems
- singular value problems
- On CCR systems, the Intel MKL is generally preferred (includes optimized BLAS and LAPACK)

www.netlib.org/lapack

Vendor BLAS

Vendor implementations of the BLAS:

AMD	ACML
Apple	Velocity Engine
Compaq	CXML
Cray	libsci
HP	MLIB
IBM	ESSL
Intel	MKL
NEC	PDLIB/SX
SGI	SCSL
SUN	Sun Performance Library

ScaLAPACK

The Scalable LAPACK library, ScaLAPACK, is designed for use on the largest of problems (typically implemented on distributed memory systems):

- subset of LAPACK routines redesigned for distributed memory MIMD parallel computers
- explicit message passing for interprocessor communication
- assumes matrices are laid out in a two-dimensional block cyclic decomposition
- On CCR systems, the Intel Cluster MKL includes ScaLAPACK libraries (includes optimized BLAS, LAPACK, and ScaLAPACK)

www.netlib.org/scalapack