

Parallel Cluster: Brief Tutorial

Mike Heroux

Department of Computer Science

CSB|SJU Systems: Beowulf

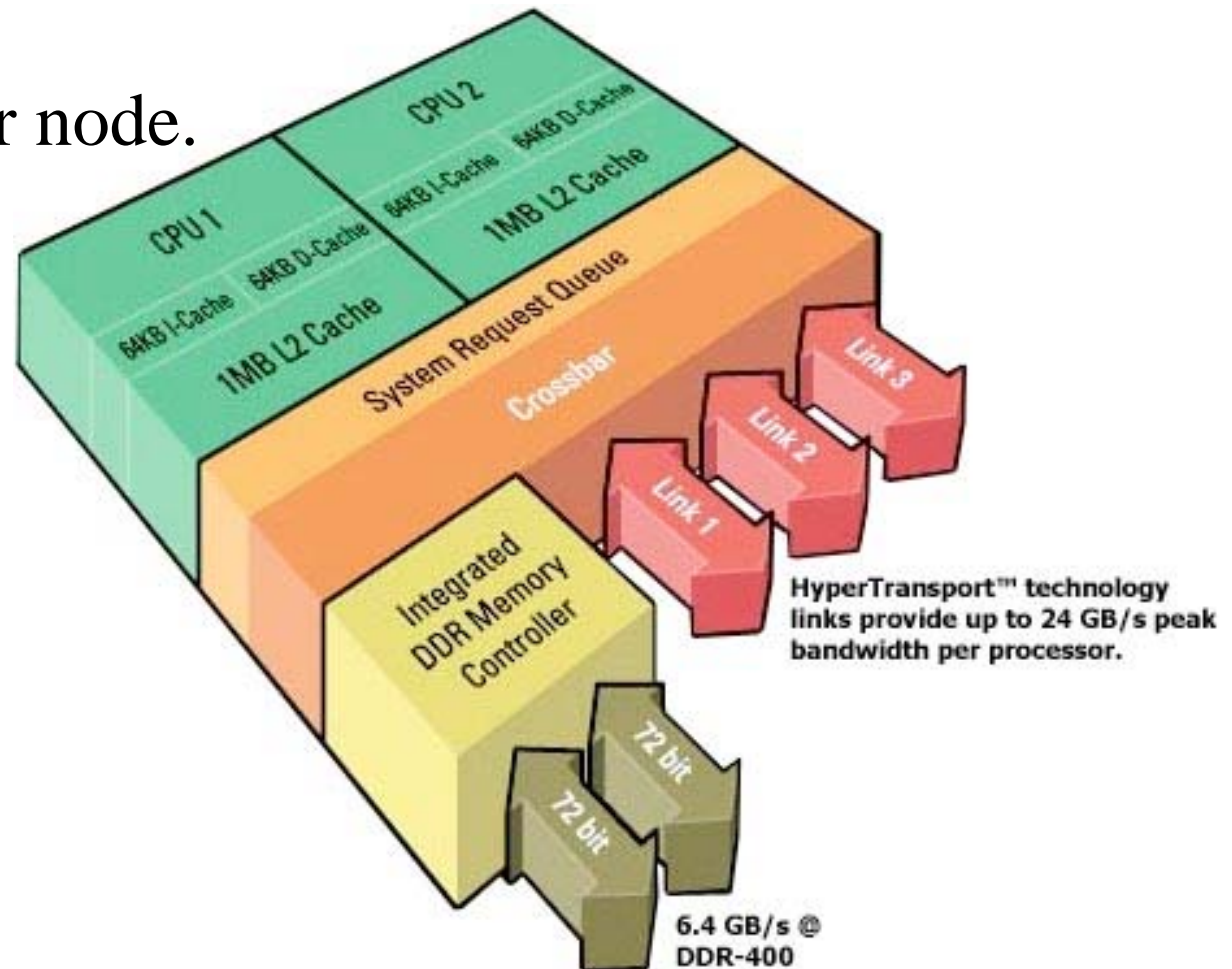
- Older system, still very useful.
- 16 identical nodes: A “more pure” MPI machine.
- Root node:
 - Has 500 GB drive
 - serves home directories to the other nodes via NFS.
- Each other node has a hard drive with an operating system on it.
- Node Specs:
 - Single Processor 32bit AMD Athlon Barton 2600 (1.9 Ghz)
 - 1 GB of 333 Mhz DDR
 - 100 mbps Ethernet
 - 32 bit Linux

CSB|SJU Systems: Melchior

- Newer system (we will use today).
- 1 master node and 6 slave nodes for a total of 7 nodes.
- Master node has a RAID 5 array with 750 GB available for data storage.
- The 750 GB file system is visible to all nodes via NFS.
- Node Specs:
 - Dual Processor Dual Core 64bit AMD Opteron 270 (2 Ghz) (4 cores total)
 - 4 GB 400 Mhz DDR (maybe increased to 6 GB soon via RAM donated by IT)
 - 1 Gbps Ethernet (second Ethernet network is planned, not implemented yet)
 - 64 bit Linux

Chip Block Architecture

- Mel Processors.
- Two of these per node.



processor : 0
vendor_id : AuthenticAMD
cpu family : 15
model : 33
model name : Dual Core AMD Opteron(tm) Processor 275
stepping : 2
cpu MHz : 2193.772
cache size : 1024 KB
physical id : 0
siblings : 2
core id : 0
cpu cores : 2
fpu : yes
fpu_exception : yes
cpuid level : 1
wp : yes
flags : fpu vme de pse tsc ...
bogomips : 4390.32
TLB size : 1024 4K pages
clflush size : 64
cache_alignment : 64
address sizes : 40 bits physical, 48 bits virtual
power management: ts fid vid ttp

processor : 2
physical id : 1
siblings : 2
core id : 0

processor : 1
vendor_id : AuthenticAMD
cpu family : 15
model : 33
model name : Dual Core AMD Opteron(tm) Processor 275
stepping : 2
cpu MHz : 2193.772
cache size : 1024 KB
physical id : 0
siblings : 2
core id : 1
cpu cores : 2
fpu : yes
fpu_exception : yes
cpuid level : 1
wp : yes
flags : fpu vme de pse tsc ...
bogomips : 4386.89
TLB size : 1024 4K pages
clflush size : 64
cache_alignment : 64
address sizes : 40 bits physical, 48 bits virtual
power management: ts fid vid ttp

processor : 3
physical id : 1
siblings : 2
core id : 1

Overview

- Parallel Processing takes on many forms:
 - Pipelining:
 - Tightly synchronized, multiple independent phases.
 - Superscalar processor:
 - Multiple instructions per CP.
 - Multiprocessor node:
 - 2 or more processors sharing memory, I/O devices, etc.
 - Multi-node machine:
 - 2 or more nodes acting together as a single machine.
- We will focus on the latter two.

Parallel Computers

- Parallel computers come in many flavors.
- Differences are due to how memory is configured.
- The most common distinction is made between *shared memory* and *distributed memory*.

Shared Memory Computers

- Shared memory means all processors can *directly* address any memory location on the *entire* machine (or on a node for hybrid systems).
 - No other processor is required for assistance.
 - Example: Multicore laptop.
- Some variations:
 - **Uniform Memory Access (UMA)**: All memory locations are accessible with the same latency. Example: Same laptop.
 - **Non-Uniform Memory Access (NUMA)**: Some memory locations are “closer” to a processor than others. Example: SGI Altix, all machines (in reality).

Distributed Memory Computers

- Distributed memory means that a processor cannot directly address all memory on the system.
- Often each processor has its own *local memory*. This memory is:
 - Directly addressable.
 - Faster to access.
 - Accessible to other processors only via help from the local processor (though not always needed).

Parallel Programming Models

- **Distributed Memory Programming (DMP) Models**
 - **Traditional Message Passing: PVM, MPI.**
 - **One-sided: SHMEM, MPI-2.**
 - **Language Extensions: UPC, Co-Array Fortran, Chapel, etc.**
- **Data Parallel**
 - **HPF.**
- **Shared Memory Programming (SMP) Models**
 - **Directives-based: OpenMP.**
 - **Explicit thread-based: Pthreads.**
 - **Taskpool API: Intel Thread Building Blocks.**
 - **Java threads (don't know much about them).**
- **Hybrid DMP/SMP Models**
 - **Pthreads/OpenMP/TBB within MPI.**

Parallel Programming Taxonomy

- SISD - Single Instruction, Single Data
 - Traditional serial computer
- SIMD - Single Instruction, Multiple Data
 - Cyber 205, Cray vector CPU, Intel SSE
- MIMD - Multiple Instruction, Multiple Data
 - Most general form of parallel computer: multiple processors executing independent code on independent data.
- SPMD - Single Program, Multiple Data
 - Special form of MIMD. Implies all processors are running same basic code but with different data.

Message Passing Overview

- Traditional Two-sided Message Passing

- Node p sends a message.
- Node q receives it.
- p and q are both involved in transfer of data.
- Data sent/received by calling library routines.



- One-sided Message Passing

- Node p puts data into the memory of node q. or
- Node p gets data from the memory of node q.
- Node q is not involved in transfer.
- Put'ing and Get'ing done by library calls.



MPI - Message Passing Interface

- The most commonly used message passing standard.
- The focus of intense optimization by computer system vendors.
- MPI-2 includes I/O support and one-sided message passing.
- The vast majority of today's scalable industrial applications run on top of MPI.
- Supports derived data types and communicators.

Shared Memory Model Overview

- All Processes share the same memory image.
- Parallelism often achieved by having processors take iterations of a for-loop that can be executed in parallel.
- OpenMP allows a user to insert special code comments that are only recognized when compiling in parallel mode.
- Pthreads requires the user to explicitly start parallel code segments via calls to the Pthreads library.

Sample Algorithm: Dot Product

```
double ddot ( const int n,  
              const double * const x,  
              const double * const y) {  
    double result = 0;  
    for (int i=0; i<n; i++) result += x[i]*y[i];  
  
    return(result);  
}
```

Sample Programs

- From melchior:
 - “cp ~mheroux/ddot.tar.gz .”
 - “tar xvzf ddot.tar.gz”
 - “cd ddot”
 - Edit (using vi or nano) ddot_serial.c:
 - “vi ddot_serial.c”

melchior.cs.csbsju.edu.146% more Makefile

CC=gcc

MPICC=mpicc

CFLAGS = -O3

OMP_CFLAGS = -fopenmp

PTHREADS_LIBS = -lpthread

MPI_LIBS = -lmpi

TARGETS = ddot_pthreads ddot_omp ddot_mpi ddot_mpi_pthreads

all: \$(TARGETS)

ddot_serial: ddot_serial.c

\$(CC) -o ddot_serial \$(CFLAGS) ddot_serial.c

ddot_pthreads: ddot_pthreads.c

\$(CC) -o ddot_pthreads \$(CFLAGS) ddot_pthreads.c \$(PTHREADS_LIBS)

ddot_omp: ddot_omp.c

\$(CC) -o ddot_omp \$(CFLAGS) \$(OMP_CFLAGS) ddot_omp.c

ddot_mpi: ddot_mpi.c

\$(MPICC) -o ddot_mpi \$(CFLAGS) ddot_mpi.c \$(MPI_LIBS)

ddot_mpi_pthreads: ddot_mpi_pthreads.c

\$(MPICC) -o ddot_mpi_pthreads \$(CFLAGS) ddot_mpi_pthreads.c \$(MPI_LIBS) \$(PTHREADS_LIBS)

clean:

rm -f *.o \$(TARGETS)

Makefile

EXAMPLE 1 - Serial DOT PRODUCT in C (ddot_serial.c)

```
#include <stdio.h>
#include <malloc.h>

/* Define constants */
/* No threads are used but makes vectors same
length as in threaded examples */
#define MAXTHRDS 4
#define MAXTRIALS 10
#define VECLen 10000000

double dotprod(int n, double * x, double * y) {
/* Define and use local variables for convenience */
int i;
double result;

result = 0.0;
for (i=0; i < n ; i++) {
    result += (x[i] * y[i]);
}
return(result);
}

/* Basic main program for serial program */

main (int argc, char* argv[]) {
    size_t i, j;
    double *x, *y, sum;
    int n=MAXTHRDS*VECLen;
/* Assign storage and initialize values*/
x = (double*) malloc(MAXTHRDS*VECLen*sizeof(double));
y = (double*) malloc(MAXTHRDS*VECLen*sizeof(double));
for (i=0; i< n;i++) {
    x[i]=1;
    y[i]=x[i]; }
for(j=0;j< MAXTRIALS;j++)
    sum = dotprod(n, x, y);
printf ("Sum = %f \n", sum);
free (x);
free (y);
return(0);
}
```

EXAMPLE 2 - OpenMP DOT PRODUCT in C (ddot_omp.c)

```
#include <stdio.h>
#include <malloc.h>

#define MAXTRIALS 10
#define MAXTHRDS 4
#define VECLLEN 10000000
double dotprod(int n, double * x, double * y) {
    int i;
    double result;
    result = 0;
#pragma omp parallel for reduction(+:result)
    for (i=0; i <n ; i++)
        result+= (x[i] * y[i]);

    return(result);
}

/* The main program initializes data and calls the dotprod()
   function. Finally, it prints the result. */

int main (int argc, char* argv[])
{
    int i,j,len;
    double *x, *y, sum;

    omp_set_num_threads(MAXTHRDS);
    len = VECLLEN*MAXTHRDS;

    x = (double*) malloc(len*sizeof(double));
    y = (double*) malloc(len*sizeof(double));

    for (i=0; i< len;i++) {
        x[i]=1;
        y[i]=x[i];
    }
    /* Perform the dotproduct */
    for (j=0;j<MAXTRIALS;j++)
        sum = dotprod(len, x, y);
    /* Print result and release storage
       */
    printf ("Sum = %f len = %d \n", sum, len);
    free (x);
    free (y);
}
```

EXAMPLE 3 - DOT PRODUCT in MPI (ddot_mpi.c)

```
#include <mpi.h>
#include <stdio.h>
#include <malloc.h>
/* Define constants */
#define MAXTRIALS 10
#define VECLEN 10000000
double dotprod(int n, double * x, double * y) {
    int i, myid;
    double result;
    result = 0.0;
    for (i=0; i < n ; i++) {
        result += (x[i] * y[i]);
    }
    return(result);
}
```

/*As before, the main program does very little computation. It does, however, make all the calls to the MPI routines. This is not a master-slave arrangement and all nodes participate equally in the work. */

```
main (int argc, char* argv[])
{
    int i,j,len=VECLEN;
    int myid, numprocs;
    double *x, *y;
    double mysum, allsum;
    double tstart, tstop;
```

```
/* MPI Initialization */
    MPI_Init (&argc,&argv);
    MPI_Comm_size (MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD,&myid);
/* Assign storage and initialize values
*/
    x = (double*) malloc(len*sizeof(double));
    y = (double*) malloc(len*sizeof(double));
    for (i=0; i< len;i++) {
        x[i]=1;
        y[i]=x[i];
    }
    tstart = MPI_Wtime();
    for (j=0;j<MAXTRIALS;j++) {
/* Call the dot product routine */
        mysum = dotprod(len, x, y);
/* After the dot product, perform
a summation of results on each node */
        MPI_Reduce (&mysum,&allsum, 1, MPI_DOUBLE,
                    MPI_SUM, 0, MPI_COMM_WORLD);
    }
    tstop = MPI_Wtime();
    if (myid == 0)
        printf ("Allsum = %f, Wall Time = %f\n", allsum, tstop-tstart);
    free (x);
    free (y);
    MPI_Finalize();
    return(0);
}
```

pthread_create function

NAME

pthread_create - create a new thread

SYNOPSIS

```
#include <pthread.h>
```

```
int pthread_create(pthread_t * thread, pthread_attr_t * attr, void * (*start_routine)(void *), void * arg);
```

DESCRIPTION

pthread_create creates a new thread of control that executes concurrently with the calling thread. The new thread applies the function start_routine passing it arg as first argument. The new thread terminates either explicitly, by calling pthread_exit(3), or implicitly, by returning from the start_routine function. The latter case is equivalent to calling pthread_exit(3) with the result returned by start_routine as exit code.

The attr argument specifies thread attributes to be applied to the new thread. See pthread_attr_init(3) for a complete list of thread attributes. The attr argument can also be NULL, in which case default attributes are used: the created thread is joinable (not detached) and has default (non real-time) scheduling policy.

RETURN VALUE

On success, the identifier of the newly created thread is stored in the location pointed by the thread argument, and a 0 is returned. On error, a non-zero error code is returned.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_exit(3), pthread_join(3), pthread_detach(3), pthread_attr_init(3).

EXAMPLE 4 - DOT PRODUCT in Pthreads (ddot_pthreads.c)

```
#include <pthread.h>
#include <stdio.h>
#include <malloc.h>
typedef struct
{
    double      *a;
    double      *b;
    double      sum;
    int         veclen;
} DOTDATA;
/* Define globally accessible variables and a mutex */
#define MAXTHRDS 4
#define VECLLEN 100
DOTDATA dotstr;
pthread_t callThd[MAXTHRDS];
pthread_mutex_t mutexsum;
void* dotprod(void *arg)
{
    /* Define and use local variables for convenience */
    int i, start, end, offset, len ;
    double mysum, *x, *y;
    offset = (size_t)arg;
    len = dotstr.veclen;
    start = offset*len;
    end = start + len;
    x = dotstr.a;
    y = dotstr.b;
    /* Perform the dot product and assign result
    to the appropriate variable in the structure. */

    mysum = 0;
    for (i=start; i < end ; i++)
    {
        mysum += (x[i] * y[i]);
    }
    /* Lock a mutex prior to updating the value in the shared
    structure, and unlock it upon updating. */
    pthread_mutex_lock (&mutexsum);
    dotstr.sum += mysum;
    pthread_mutex_unlock (&mutexsum);
    pthread_exit((void*)0);
}

/*The main program creates threads that do all the work and
then print out result upon completion. Before creating the
threads, the input data is created. Since all threads update a
shared structure, we need a mutex for mutual exclusion. The
main thread needs to wait for all threads to complete, it waits
for each one of the threads. We specify a thread attribute
value that allow the main thread to join with the threads it
creates. Note also that we free up handles when they are no
longer needed. */
void main (int argc, char* argv[])
{
    size_t i;
    double *a, *b;
    int status;
    pthread_attr_t attr;
    /* Assign storage and initialize values*/
    a = (double*) malloc(MAXTHRDS*VECLLEN*sizeof(double));
    b = (double*) malloc(MAXTHRDS*VECLLEN*sizeof(double));
    for (i=0; i< VECLLEN*MAXTHRDS;i++)
    { a[i]=1;
      b[i]=a[i]; }
    dotstr.veclen = VECLLEN;
    dotstr.a = a;
    dotstr.b = b;
    dotstr.sum=0;
    pthread_mutex_init(&mutexsum,NULL);
    /* Create threads to perform the dotproduct */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
    for(i=0;i< MAXTHRDS;i++)
    { /** Each thread works on a different set of data.
      ** The offset is specified by 'i'. The size of
      ** the data for each thread is indicated by VECLLEN.*/
      pthread_create( &callThd[i], &attr, dotprod, (void *)i);
    }
    pthread_attr_destroy(&attr);
    /* Wait on the other threads */
    for(i=0;i< MAXTHRDS;i++)
    {pthread_join( callThd[i], (void **)&status);}
    /* After joining, print out the results and cleanup */
    printf ("Sum = %f \n", dotstr.sum);
    free (a);
    free (b);
    pthread_mutex_destroy(&mutexsum);
    pthread_exit (0);
}
```

Hybrid DMP/SMP Models

- Many applications exhibit a coarse grain parallel structure and a simultaneous fine grain parallel structure nested within the coarse.
- Many parallel computers are essentially clusters of SMP nodes.
 - SMP parallelism is possible within a node.
 - DMP is required across nodes.
- Compels us to consider programming models where, for example, MPI runs across nodes and OpenMP runs within nodes.

EXAMPLE 5 - DOT PRODUCT in MPI with Pthreads (ddot_mpi_pthreads.c)

```
#include <mpi.h>
#include <pthread.h>
#include <malloc.h>
typedef struct
{
    double      *a;
    double      *b;
    double      sum;
    int         veclen;
    int         numthrds;
} DOTDATA;
/* Define globally accessible variables and a mutex */
#define MAXTHRDS 4
#define VECLLEN 100
    DOTDATA dotstr;
    pthread_t callThd[MAXTHRDS];
    pthread_mutex_t mutexsum;
void* dotprod(void *arg)
{
    /* Define and use local variables for convenience */
    int i, start, end, mythrd, len, numthrds, myid;
    double mysum, *x, *y;
    /* The number of threads and nodes defines the beginning
    and ending for the dot product; each thread does work
    on a vector of length VECLLENGTH. */
    mythrd = (int)arg;
    MPI_Comm_rank (MPI_COMM_WORLD, &myid);
    numthrds = dotstr.numthrds;
    len = dotstr.veclen;
    start = myid*mythrd*len + mythrd*len;
    end = start + len;
    x = dotstr.a;
    y = dotstr.b;
    /* Perform the dot product and assign result
    to the appropriate variable in the structure. */
    mysum = 0;
    for (i=start; i < end ; i++)
    {
        mysum += (x[i] * y[i]);
    }
    /* Lock a mutex prior to updating the value in the
    structure, and unlock it upon updating. */
    pthread_mutex_lock (&mutexsum);
    dotstr.sum += mysum;
    pthread_mutex_unlock (&mutexsum);
    pthread_exit((void*)0);
}

void main (int argc, char* argv[])
{
    size_t i;
    int myid, numprocs, len=VECLLEN;
    int numpl, numthrds;
    double *a, *b;
    double nodesum, allsum;
    int status;
    pthread_attr_t attr;
    /* MPI Initialization */
    MPI_Init (&argc,&argv);
    MPI_Comm_size (MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD,&myid);
    /* Assign storage and initialize values*/
    numthrds=MAXTHRDS;
    a = (double*) malloc(numthrds*len*sizeof(double));
    b = (double*) malloc(numthrds*len*sizeof(double));
    for (i=0; i<numthrds*len; i++) {
        a[i]=1; b[i]=a[i];}
    dotstr.veclen = len;
    dotstr.a = a; dotstr.b = b; dotstr.sum=0;
    dotstr.numthrds=MAXTHRDS;
    /* Create thread attribute to specify that the main thread
    needs to join with the threads it creates. */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
    /* Create a mutex */
    pthread_mutex_init (&mutexsum,NULL);
    /* Create threads within this node to perform dotproduct */
    for(i=0;i< numthrds;i++){
        pthread_create( &callThd[i], &attr, dotprod, (void *)i);}
    /* Release the thread attribute handle - no longer needed */
    pthread_attr_destroy(&attr);
    /* Wait on the other threads within this node */
    for(i=0;i< numthrds;i++){
        pthread_join( callThd[i], (void **)&status); }
    nodesum = dotstr.sum;
    /* After the dotprod, perform a sum of results on each node */
    MPI_Reduce (&nodesum, &allsum, 1, MPI_DOUBLE,
                MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) printf ("Allsum = %f \n", allsum);
    MPI_Finalize();
    free (a); free (b);
    pthread_mutex_destroy(&mutexsum);
    exit (0);
}
```

EXAMPLE 6 - DOT PRODUCT in MPI OpenMP (ddot_mpi_omp.c)

```
#include <mpi.h>
#include <stdio.h>
#include <malloc.h>
/* Define constants */
#define MAXTRIALS 10
#define VECLEN 10000000
double dotprod(int n, double * x, double * y) {
    int i, myid;
    double result;
    result = 0.0;
    #pragma omp parallel for reduction(+:result)
    for (i=0; i < n ; i++) {
        result += (x[i] * y[i]);
    }
    return(result);
}
/*As before, the main program does very little computation.
It does, however, make all the calls to the MPI routines.
This is not a master-slave arrangement and all nodes
participate equally in the work. */

main (int argc, char* argv[])
{
    int i,j,len;
    int myid, numprocs;
    double *x, *y;
    double mysum, allsum;
    double tstart, tstop;

    /* MPI Initialization */
    MPI_Init (&argc,&argv);
    MPI_Comm_size (MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD,&myid);
    omp_set_num_threads(MAXTHRDS);
    len = VECLEN*MAXTHRDS;

    /* Assign storage and initialize values
    */
    x = (double*) malloc(len*sizeof(double));
    y = (double*) malloc(len*sizeof(double));
    for (i=0; i< len;i++) {
        x[i]=1;
        y[i]=x[i];
    }
    tstart = MPI_Wtime();
    for (j=0;j<MAXTRIALS;j++) {
        /* Call the dot product routine */
        mysum = dotprod(len, x, y);
        /* After the dot product, perform
        a summation of results on each node */
        MPI_Reduce (&mysum,&allsum, 1, MPI_DOUBLE,
                    MPI_SUM, 0, MPI_COMM_WORLD);
    }
    tstop = MPI_Wtime();
    if (myid == 0)
        printf ("Allsum = %f, Wall Time = %f \n", allsum, tstop-tstart);
    free (x);
    free (y);
    MPI_Finalize();
    return(0);
}
```