Parallelization with OpenMP and MPI
A Simple Example (C)

Dieter an Mey, Thomas Reichstein

October 26, 2007

1 Introduction

The main aspects of parallelization using MPI (Message Passing Interface) on one hand and OpenMP directives on the other hand shall be shown by means of a toy program calculating π.

Parallelization for computer systems with distributed memory (DM) is done by explicit distribution of work and data on the processors by means of message passing.

Parallelization for computer systems with shared memory (SM) means automatic distribution of loop partitions on multiple processors, or the explicit distribution of work on the processors with compiler directives and runtime function calls (OpenMP).

MPI programs also run on shared memory systems, whereas OpenMP programs do not normally run on distributed memory machines (one exception is Intel's Cluster OpenMP)

The combination of a coarse-grained parallelization with MPI and an underlying fine-grained parallelization of the individual MPI-tasks with OpenMP is an attractive option to use a maximum number of processors efficiently. This method is known as hybrid parallelization.

2 Problem definition, serial program and automatic parallelization

π can be calculated as an integral:

$$\pi = \int_o^1 f(x)\,\mathrm{dx}\ ,\ \textit{with}\ f(x) = \frac{4}{(1+x)^2} \tag{2.1}$$

This integral can be numerically approximated through a quadrature method (rectangle method):

$$\pi = \frac{1}{n}\sum_{i=1}^{1} f(x_i)\ ,\ \textit{with}\ x_i = \frac{(i-\frac{1}{2})}{n}\ \textit{for}\ i = 1, ..., n \tag{2.2}$$

The following serial program allows to vary the number of nodes n, until entering zero stops the execution.

```c
/**********************************************************************
* *
* compute pi by intergrating f(x) = 4/(1 + x**2) *
* *
* Variables: *
* *
* pi the calculated result *
* n number of point of integration *
* x midpoint of each rectangle's interval *
* f function to integrate *
* sum, pi area of rectangles *
* tmp temporary scratch space for global summation *
* i loop index *
* *
**********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.141592653589793238462643383279502 9L
double f(double a) {
    return (double)4.0/((double)1.0+(a*a));
}
int main(int argc, char *argv[])
{
    unsigned long n,i;
    double h,pi,sum,x;
    for (;;) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%u",&n);
        if (n==0)
            break;
        h = ((double)1.0)/(double)n;
        sum = 0.0;
        for(i=1;i<=n;i++) {
           x = h*((double)i-(double)0.5);
           sum += f(x);
        }
        pi = h*sum;
        printf(" pi is approximatly: %.16f Error is: %.16f\n",
        pi,fabs(pi-PI));
    }
    return EXIT_SUCCESS;
}
```

Entering a 10 followed by a 0, the output looks like the following :

```
Enter the number of intervals: (0 quits)
pi is approximately: 3.1415926535981615      Error is: 0.0000000000083684
Enter the number of intervals: (0 quits)
```

The approximated solution is being compared with a solution accurate to the 25th position.

In this simple example the function f(x) being integrated is quit simple and parallelization only pays off when the number of nodes is quite high. Parallelization of an integral with a numerically more expensive function would be a lot more profitable.

The program core, to be parallelized, mainly consists of an inner loop, in which the sum of the values of the function f(x) at the nodes is calculated.

```
h = ((double)1.0)/(double)n;
sum = 0.0;
for(i=1;i<=n;i++) {
    x = h*((double)i-(double)0.5);
    sum += f(x);
}
pi = h*sum;
```

The evaluation of the individual loop iterations have to be distributed to several processors.

In this simple case, the compiler is usually able to automatically parallelize this loop for a shared memory machine. The only problem arises through the recursive use of the variable **sum**, which is read and modified with each loop pass, so that every cycle depends on the previous one. Using the associativity of the summation, parallelization in this case is possible. These rounding errors usually differ from those caused by serial execution. With the the Sun Compiler you therefore have to use both the **-autopar** and the **-reduction** options.

3 Parallelization for distributed memory through Message Passing with MPI

3.1 Preliminary note

Processes with their own address space have to cooperate in order to utilize parallel machines with distributed memory. To ease the communication between separate processes, the MPI Message Passing Library was developed. The sending and receiving of messages is achieved through standardized subroutine calls, so that an MPI program is portable to all machines for which an MPI Library is available. With public domain software packages **mpich2** or **OpenMPI**, every machine that supports the tcp/ip protocol can be used.

MPI programs typically follow the SPMD programming style (Single Program Multiple Data). All involved MPI-processes, execute the same binary program, and after initialization with **MPI_Init**, every process gets the total number of parties involved with the call **MPI_Comm _size** and its own identification by calling **MPI_Comm_rank**. The task with identification zero then usually takes charge ("Master").

3.2 MPI_Send and MPI_Recv

Here, a so called worker farm is an obvious approach to paralellize this simple example. The master process takes care of the input and output leaving just the evaluation of the inner loop to the other processes. Initially the master-process has to provide all its workers with the necessary data, in this case just the value **n** and towards the end the worker-processes have to send the partial results to the master so that the master can combine them to the overall result.

At the beginning, the master sends the value **n** with **MPI_Send** to all workers. The workers in return have to receive the data with **MPI_Recv**.

The partitioning of the loop indices to all tasks was made in cyclic fashion:

```
for (i = myid + 1; i <= n; i += numprocs)
```

Another option would be to divide the loop iterations into chunks:

```
chunksize = ( n + ntasks     1 ) / ntasks;
for (i = myid*chunksize+1; i <= min(n,(myid+1)*chunksize); i += 1 ) . .
```

Here, the master takes part in the computation, which is not necessarily always the case. Finally each worker sends its partial sum **mypi** to the master for collection and adding up the final result.

All tasks leave with **MPI_Finalize** the MPI environment at program end.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#define PI 3.1415926535897932384626433832795029L
double f(double a) {
    return (double)4.0/((double)1.0+(a*a));
}
#define MTAG1 1
#define MTAG2 2
int main(int argc, char *argv[])
{
    int n, myid, numprocs, i, islave;
    double mypi, pi, h, sum, x;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    n = 0;
    for (;;) {
        if (myid == 0) {
            printf("Enter␣the␣number␣of␣intervals:␣(0␣quits)␣"); scanf("%d",&n);
            for (islave=1;islave<numprocs;islave++) {
                MPI_Send( &n, 1, MPI_INTEGER, islave, MTAG1, MPI_COMM_WORLD);
            }
        } else {
            MPI_Recv (&n, 1, MPI_INTEGER, 0, MTAG1, MPI_COMM_WORLD,&status);
        }
        if (n == 0)
            break;
        else {
            h = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double)i - 0.5);
                sum += f(x);
            }
            mypi = h * sum;
            if (myid != 0) {
                MPI_Send (&mypi, 1, MPI_DOUBLE_PRECISION,0,MTAG2,MPI_COMM_WORLD);
            } else {
                pi = mypi;
                for (islave=1;islave<numprocs;islave++) {
                    MPI_Recv (&mypi, 1, MPI_DOUBLE_PRECISION, islave,
                    MTAG2, MPI_COMM_WORLD, &status);
                    pi += mypi;
                }
                printf("pi␣is␣approx.␣%.16f,␣Error␣is␣%.16f\n", pi, fabs(pi - PI));
            }
        }
    }
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

3.3 MPI_Bcast and MPI_Reduce

The frequent operations “one sends to all“ and ”all send to one“ can be implemented more elegantly through the special MPI calls **MPI_Bcast** and **MPI_Reduce** respectively. These calls are designed in such a way, that to differentiate between sender and receiver no control structures have to be programmed, just the so called root-parameter has to be set.

Attention: When using the reduction function **MPI_Reduce** in conjunction with the parameter **MPI_SUM** its not warranted that you allways receive a numerically identical result, since the MPI library takes advantage of the associativity of the summation. This can result in differend rounding errors.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#define PI 3.1415926535897932384626433832795029L
#define MTAG1 1
#define MTAG2 2
double f(double a) {
    return (double)4.0/((double)1.0+(a*a));
}
int main(int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    n = 0;
    for (;;) {
        if (myid == 0) { printf("Enter␣...␣"); scanf("%d",&n); }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;
    else {
        h = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double)i - 0.5);
            sum += f(x);
        }
        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        if (myid == 0) { printf("pi␣is␣approx...␣\n", pi, fabs(pi - PI));
        }
    }
}
MPI_Finalize();
return EXIT_SUCCESS;
}
```

4 Parallelization for Shared Memory through OpenMP Directives

4.1 Preliminary note

For shared memory programming OpenMP is the de facto standard. The OpenMP API is defined for Fortran, C and C++, it comprises of compiler directives, runtime routines, and environment variables.

At the beginning of the first parallel region of an OpenMP program (these are the program parts enclosed in the **parallel** directive), several lightweight processes sharing one address space, so called *Threads*, are started. These threads execute the parallel region redundantly until they reach a so called *Worksharing Construct*, in which the arising work (usually Fortran **DO** , or C/C++ **for** loops) is divided among the *Threads*.
Normally *Threads* can access all data (shared data) likewise.
Attention: In case several *Threads* modify the same shared data, access to it has to be protected in *Critical Regions* (the program part followed/enclosed by the **critical** directive).
Private Data in which the individual *Threads* store their temporary data can be used as well. Local data of subprograms, which are called inside of parallel regions, are private too, because they are put on the stack. As a consequence, they do not maintain their contents from one call to the next!

4.2 The parallel and the for Directives

Again, the inner loop shall be parallelized, here with OpenMP directives.

```
h = ((double)1.0)/(double)n;
sum = 0.0;
for(i=1;i<=n;i++) {
    x = h*((double)i-(double)0.5);
    sum += f(x);
}
pi = h*sum;
```

The first step is to enclose the inner loop with the **parallel** directive. To prevent that all threads execute this loop redundantly, the **for** directive is added to the loop in order to distribute the loop iterations to all processors (worksharing).

Since by default all variables are accessible by all threads (shared), the exceptions have to be taken care of. A first candidate for privatization is the loop index **i**. If the loop iterations shall be distributed, the loop index has to be private. This is realized through a **private** clause of the **parallel** directive. As a second candidate for privatization there is the variable **x**, which is used to temporally store the node of the quadrature formula. This happens independently for each loop iteration and the variable contents is not needed after the loop.

With the usage of the summation variable **sum** it gets more complicated. On the one hand, the variable is used by all threads equally to calculate the sum of the quadrature formula, on the other hand it is set to zero prior to the loop and is needed after the loop to calculate the final solution. Would the variable be **shared**, the following problem could arise: a thread reads the value of **sum** from memory and puts it in the cache to add up his newly calculated value of **f(x)**. But before the sum can be written back to memory, another thread may read **sum** from memory to also add up a new function value. This way the contribution of the first thread may be lost.

This situation can be avoided, if only the function values are computed in parallel and stored in an auxiliary array **fx** and the summation is processed by the master thread only.

```
#include <omp.h>
...
int main(int argc, char *argv[])
{
    ...
    for (;;) {
        printf("Enter␣the␣number␣of␣intervals:␣(0␣quits)␣"); scanf("%u",&n);
        if (n==0) break;
        fx = (double *) malloc(n*sizeof(double));
        h = ((double)1.0)/(double)n;
        sum = 0.0;
#       pragma omp parallel private(i,x) shared(h,fx,n)
        {
#           pragma omp for
            for(i=1;i<=n;i++) {
                x = h*((double)i-(double)0.5);
                fx[i-1] = f(x);
            }
        }
        for(i=0;i<n;i++) {
            sum += fx[i];
        }
        pi = h*sum;
        printf("␣pi␣is␣approx.:␣%.16f␣Error␣is:␣%.16f\n",pi,fabs(pi-PI));
    }
    return EXIT_SUCCESS;
}
```

The array **fx** can confidently be declared **shared** with the corresponding clause of the parallel directive (this also is the default), since the individual threads use different loop indexes **i** and thus access different components of the array **fx**.

4.3 The critical Directive

The second solution makes use of the possibility to protect code sequences in critical regions, in which several threads modify shared variables. Critical regions are segments of code which can only be executed by a single thread at a time.

```
int main(int argc, char *argv[])
{
    ...
    h = ((double)1.0)/(double)n;
    sum = 0.0;
#   pragma omp parallel
    {
#       pragma omp for private(i,x)
        for(i=1;i<=n;i++) {
            x = h*((double)i-(double)0.5);
#           pragma omp critical
            sum += f(x);
        }
    }
    pi = h*sum;
    ...
}
```

This version however involves quite some overhead, because it introduces a synchronization with every iteration of the inner loop.

The next version introduces an additional private variable, in which the individual threads sum up their contributions. The total sum is then computed in a critical region after the parallel loop.

```
int main(int argc, char *argv[])
{
    double sum_local;
    ...
    h = ((double)1.0)/(double)n;
    sum = 0.0;
#   pragma omp parallel private(i,x,sum_local)
    {
        sum_local = 0.0;
#       pragma omp for
        for(i=1;i<=n;i++) {
            x = h*((double)i-(double)0.5);
            sum_local += f(x);
        }
#       pragma omp critical
        sum += sum_local;
    }
    pi = h*sum;
    ...
}
```

This solution finally executes with a reasonable speedup.

4.4 The reduction clause

Exactly for this case there exists - analogous to the reduction function in MPI - a **reduction** clause of the **for** directive. Through its usage, the parallel program gets pleasantly short and manageable.

Attention: When the reduction clause is used with the + operator it is not warranted that numerically identical solutions are generated everytime, because the associativity of the summation is utilized. Different rounding errors can occur as a result.

```
int main(int argc, char *argv[])
{
    ...
    h = ((double)1.0)/(double)n;
    sum = 0.0;
#   pragma omp parallel private(i,x)
    {
#       pragma omp for reduction(+:sum)
        for(i=1;i<=n;i++) {
            x = h*((double)i-(double)0.5);
            sum += f(x);
        }
    }
    pi = h*sum;
    ...
}
```

This version can be programmed even more concise with just one directive.

```
int main(int argc, char *argv[])
{
    ...
    h = ((double)1.0)/(double)n;
    sum = 0.0;
#   pragma omp parallel for private(i,x) reduction(+:sum)
    for(i=1;i<=n;i++) {
        x = h*((double)i-(double)0.5);
        sum += f(x);
    }
    pi = h*sum;
    ...
}
```

4.5 The single and the barrier Directives

Yet, the usage of OpenMP does not limit itself just on parallelizing (inner)loops. In the following example the entire executable part of the program is enclosed in the parallel region.

```
int main(int argc, char *argv[])
{
    ...
#   pragma omp parallel private (i,x)
    {
        for (;;) {
#           pragma omp single
            {
                printf("Enter␣the␣number␣of␣intervals:␣(0␣quits)␣");
                scanf("%u",&n);
            }
            if (n==0)
                break;
            h = ((double)1.0)/(double)n;
            sum = 0.0;
#           pragma omp barrier
#           pragma omp for private(i,x) reduction(+:sum)
            for(i=1;i<=n;i++) {
                x = h*((double)i-(double)0.5);
                sum += f(x);
            }
#           pragma omp single
            {
                pi = h*sum;
                printf("␣pi␣is␣approx.:␣%.16f␣Error␣is:␣%.16f\n",pi,fabs(pi-PI));
            {
        }
    }
    return EXIT_SUCCESS;
}
```

Therefore read and writes are enclosed in the **single** directive, causing execution just by a single thread, the first one to reach this point in the program code. The **single** directive contains an implicit barrier at the end, so that when the if statement is reached, all threads use the current value of the just read variable **n**. The evaluation of the stride **h** and the initialization of the summation variable **sum** is done by this simple thread too. The **barrier** which is included in the **single** directive before the inner loop is quite important! Without the barrier, it would be possible that an “early” thread already has delivered its contribution the summation, when a “later” thread puts the first summation variable to zero. Thus, the contribution of the “early” thread would be lost.

4.6 Orphaning

The next OpenMP-program version explores the possibility of orphaning. Directives inside of a parallel region do not necessarily have to be included in the same program module. They also can reside in subprograms which are called from inside a parallel region.

```
void cal_pi(long n, double *pi)
{
    long i;
    double a, x;
    static double sum, h;

#   pragma omp single
    {
        h = ((double)1.0)/(double)n;
        sum = 0.0;
    }

#   pragma omp for private(i,x) reduction(+:sum)
    for(i=1;i<=n;i++) {
        x = h*((double)i-(double)0.5);
        sum += f(x);
    }

#   pragma omp single
    {
        *pi = h*sum;
    }
    return;
}

int main(int argc, char *argv[])
{
    ...
#   pragma omp parallel
    {
        for (;;) {
#           pragma omp single
            {
                printf("Enter␣...:␣(0␣quits)␣"); scanf("%u",&n);
            }
            if (n==0) break;
            cal_pi(n,&pi);
#           pragma omp single
            {
                printf("␣pi␣is␣approx.:␣%.16f␣Error␣is:␣%.16f\n",pi,fabs(pi-PI));
            }
        }
    }
    ...
}
```

Hence the evaluation of the inner loops including the pre- and postprocessing has been sourced out to the subprogram **calc_pi**. The main program now just consists of the input and output parts and the outer loop. Here one has to bear in mind that usually all local variables of such a subprogram are automatically private since they are allocated on the stack. Otherwise multiple threads could not concurrently pass through the same sub program, since they then would destroy each others local variables (thread safety!). In this case however the variables **h** and **sum** are supposed to be used shared! So they have to be explicitly declared static. In Fortran this can be done with **COMMON** blocks, through modules, or with the **SAVE** attribute. In C variables have to be declared **static**.

4.7 The omp_get_thread_num and omp_get_num_threads functions

The last program version suggests that one is not tied to the parallelization of loops when programming with OpenMP. On the contrary, through the usage of the function calls **omp_get _thread_num**, and **omp_get_num_threads**, which provide the thread-identification and the number of active threads resp., one can develop a program which reminds of the MPI version.

```
int main(int argc, char *argv[])
{
    ...
#   pragma omp parallel private (i,x)
    {
        for (;;) {
#           pragma omp single
            {
                printf("Enter␣the␣number␣of␣intervals:␣(0␣quits)␣");
                scanf("%u",&n);
            }
            if (n==0)
                break;
            h = ((double)1.0)/(double)n;
            sum = 0.0;
#           pragma omp barrier
#           pragma omp for private(i,x) reduction(+:sum)
            for(i=1;i<=n;i++) {
                x = h*((double)i-(double)0.5);
                sum += f(x);
            }
#           pragma omp single
            {
                pi = h*sum;
                printf("␣pi␣is␣approx.:␣%.16f␣Error␣is:␣%.16f\n",pi,fabs(pi-PI));
            }
        }
    }
    return EXIT_SUCCESS;
}
```

5 Hybrid Parallelization using MPI and OpenMP

Parallelization with MPI on the top (coarse-grained) layer and with OpenMP on the bottom (fine-grained) layer can easily be combined. Thereby individual MPI-tasks are parallelized with OpenMP, or viewed from another angle, the MPI-library calls take place in the serial regions.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <omp.h>
#define PI 3.1415926535897932384626433832795029L
#define MTAG1 1
#define MTAG2 2

double f(double a) { return (double)4.0/((double)1.0+(a*a)); }

int main(int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    n = 0;
    for (;;) {
        if (myid == 0) { printf("Enter␣...:␣(0␣quits)␣"); scanf("%d",&n); }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
            break;
        else {
            h = 1.0 / (double) n;
            sum = 0.0;
#           pragma omp parallel for reduction(+:sum) private(i,x)
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double)i - 0.5);
                sum += f(x);
            }
            mypi = h * sum;

            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

            if (myid == 0) { printf("...\n", pi, fabs(pi - PI)); }
        }
    }
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

In this simple example a single OpenMP directive has to be introduced into the MPI version, to demonstrate a valid hybrid program.