Yet Another MPI Tutorial

Georg Bisseling

Yet Another MPI Tutorial
by Georg Bisseling

Copyright © 2005,2006,2007 Georg Bissling

Table of Contents

List of Tables

Introduction

This is another MPI tutorial, but one that presumes that you are already quite familiar with MPI in the sense that you managed to get some test programs to work on your computer(s). Installation problems etc. are not covered. Furthermore we will assume that you already wrote some MPI programs yourself and were surprised, confused or disappointed by varying amounts.

To get more information about the MPI standards please have a look at http://www.mpi-forum.org/.

Updated versions of this tutorial might be available at http://www.bisseling.de/georg/ and may be anounced in the news group news:comp.parallel.mpi.

It is really important to understand that on one hand there is a MPI standard that tells how a MPI implementation should behave and how a correct MPI program looks like and on the other hand there are several MPI implementations. It's paperware vs. software. Fortunately not only the standards are open source, there are even open source MPI implementations.

The two of them are MPICH2 http://www-unix.mcs.anl.gov/mpi/mpich/ and OpenMPI http://www.open-mpi.org/.

You may find software packets that are tailored to be built with predecessors of these two implementations, namely MPICH http://www-unix.mcs.anl.gov/mpi/mpich1/ and LAM MPI http://www.lam-mpi.org/.

Probably at least some of them were included in your favorite Linux distribution.

For this tutorial I used the software tool chain provided by Intel: compiler, MPI library and the Intel trace analyzer for performance analysis. Even if you prefer to use the gcc compiler and MPICH2 or mpich then you can still use the trace analyzer.

Most if not all of the timings were done on a quad core SMP box (Intel Q6600) running OpenSuSE 10.2.

I first started to get involved with MPI in 2000 when I got a new job in a project that aimed at implementing the whole MPI-2 standard. Although I did learn quite a bit about one could implement MPI and how one should better not, I have to admit that I lack the application programmers attitude to MPI.

When it comes to tremendous amounts of Fortran code that do some weather forcasts, then I will never get that in the right perspective, I'm afraid. But I saw some questions in the news group comp.parallel.mpi that made me think that several problems could be understood and demonstrated using very small examples.

I decided to use the calculation of the Mandelbrot set, or Apfelmännchen as we say in Germany, as

an example. There are literaly thousands of pages dealing with the Mandelbrot set, so it isn't necessary to repeat an explanation of that here.

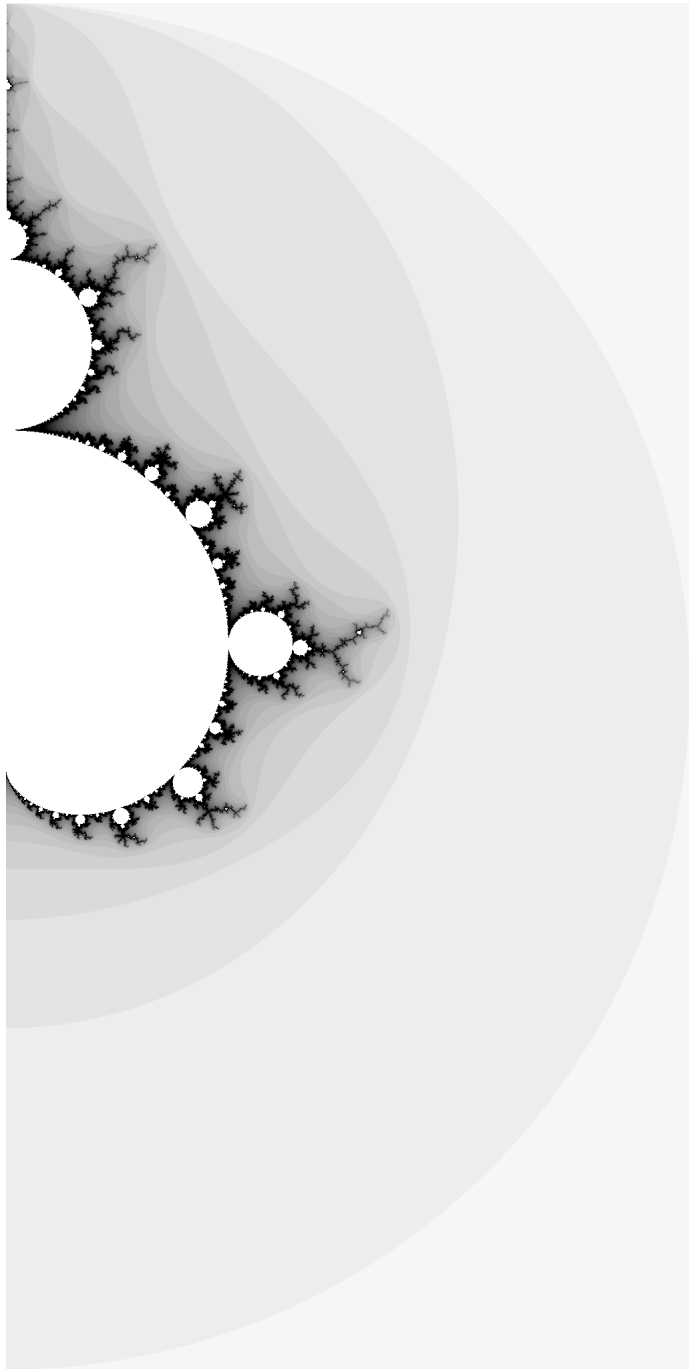
The nice properties of this problem are that the result of the Mandelbrot calculation can be represented in a bitmap picture which makes it easy and pleasing to check the correctness of the result and that the calculation of each single pixel is entirely independent of all other pixels, so that the calculation can be partitioned in multiple ways.

The tutorial will start with a serial program and convert that step by step into a parallel program thereby demonstrating several decisions that you have to make along the way, what problems can arise and how to solve them. Most of these problems are not constructed but I ran into them right away. Which is funny since the parallelization of the calculation of the Mandelbrot set should be trivial.

Several ways how to parallelize the program will be shown along with some pictures generated with the help of the Intel Trace Analyzer and Collector to examine the performance characteristics of a particular program.

In the end we will even use OpenMP to parallelize the program. Just to show that there is more to parallel programing than MPI.

Figure 1. To whet your appetite



Chapter 1. Starting with a Serial Program

We start with the serial program `apple_serial.c` shown in Appendix C. The invariant part of all the examples is in `invar.h` and `invar.c` (refer to Appendix A and Appendix B). So apple_serial.c is just the driver program.

One really interesting part in invar.c is the the routine `writeOut`. This routine can write the result of the calculation out to disk so that we can check the result as a picture. I use the ImageMagick tools for that. Check out http://www.imagemagick.org/, but your favorite Linux distribution might already have installed that on your harddisk anyway.

When I run that on a 2.4 GHz Core 2 machine then I get:

```
> time ./apple_serial
RE_START =                                        -2
RE_STOP  =                                         2
IM_START =                                         0
IM_STOP  =                                         2
XPIX     =                                      2048
YPIX     =                                      1024
MAXITER  =                                      4096
    1023
real    0m4.937s
user    0m4.868s
sys     0m0.028s
```

We have a point to start now: a single C2-2.4GHz-CPU can do it in less than 5 seconds - for this parameters that is.

Please notice that we only calculate the upper half of the Mandelbrot set because it is symmetric around the real axis. You do not get a speed up of two for free so often.

Now we set the environment variable WRITEOUT to save the results in an image and we see

```
> WRITEOUT=1 time ./apple_serial
...
Writing result to file "apple_serial_1p_size2048x1024.gray".
```

at the end of the output. Now start ImageMagick with **`display -depth 16 -size 2048x1024 apple_serial_1p_size2048x1024.gray`**

What do we see? Nothing, that's correct. We stored into a 16 bit gray scale format, but used less than the first few bits of it, which looks pretty black. Choose "Enhance" and then "Equalize". In the introduction we have this picture (rotated to fit the page and negated to save the ink: Figure 1.

Okay. We have now a serial program that seems to produce a reasonable result and we know how much time it consumes.

Now we take the next step. We introduce some MPI calls into the program but we let it still run with one process. The result is `apple_pseudo_mpi.1.c` (Appendix D). We start it like this:

```
>time ./apple_pseudo_mpi.1
...
real    0m4.956s
user    0m4.848s
sys     0m0.028s
```

Wow! No overhead for introducing MPI! The truth is that we left out the start script that is (most often) necessary to start a MPI program on a parallel machine like a cluster. Depending on the MPI (and machine) that you use it might be impossible to start the program standalone. So assuming that MPI is already properly setup we use the command mpiexec to start the same program with one process:

```
> time mpiexec -np 1 ./apple_pseudo_mpi.1
...
real    0m5.155s
user    0m0.144s
sys     0m0.048s
```

Still next to no overhead. That may differ for you setup and it will for sure differ for higher processor counts or on a cluster.

Just for the fun of it try **`time mpiexec -np 2 ./apple_pseudo_mpi.1`**. This should take double the time on a single CPU machine and the same time on a machine with two or more CPUs. What happened? We had a MPI job consisting of two processes without any useful communication. At least no communication that was programmed by us. Both processes exercised the whole calculation.

The next step is to consider how the calculation could be distributed among the processes. I suggest a very simple scheme where the calculation is distributed statically row by row. By saying *statically* I mean that given the amount of rows to calculate and the number of processes the assignment is totally determined. We will not have any protocol used to assign the work.

The scheme will look like this:

Table 1-1. Static Scheduling

Row Number	2 processes, working root	3 processes, working root	3 processes, idle root
0	rank 0	rank 0	rank 1
1	rank 1	rank 1	rank 2
2	rank 0	rank 2	rank 1
3	rank 1	rank 0	rank 2
4	rank 0	rank 1	rank 1
5	rank 1	rank 2	rank 2

The preliminary result of this effort is `apple_pseudo_mpi.2` (Appendix E). Each process uses the

macro `MYROW` to determine if it should calculate the given row. This macro itself and some other code is dependant on the definition of another macro `IDLEROOT`. If this macro is defined then the process with rank 0, often called root, will not participate in the calculation. This will help us later on. Notice that the makefile produces two versions of all programs that behave differently depending on the definition of `IDLEROOT`.

When I run it with two and four processes on a quad core machine I get:

```
> time mpiexec -np 2 ./apple_pseudo_mpi.2
...
real    0m2.832s
user    0m0.112s
sys     0m0.012s

> time mpiexec -np 4 ./apple_pseudo_mpi.2
...
real    0m1.647s
user    0m0.100s
sys     0m0.024s
```

You see that this scales very well. Both processes consume about half of the time that was spend in the serial version and we did not introduce substantial additional overhead.

Notice that the command line arguments are parsed in both processes. We will get rid of that in the next version. But try to understand what happens.

Although this program does really exercise the computation that we want to have done it is really of no use: Each rank will calculate a part of the result, but the result is not gathered in one place. You can check that by letting it write a file as we did with `apple_serial.c`. It will contain nothing if `IDLEROOT` was set and only the lines computed by rank 0 otherwise.

The next step is, as you might expected, `apple_pseudo_mpi.3` (refer to Appendix F). But it is only a small step. The command line is parsed only at rank 0 and the results are broadcasted to the other ranks. If the parsing fails for some reason, then all ranks are told to exit in an orderly fashion. But still the result of the calculation is not gathered.

You might have noticed that only the changes between the programs are decorated with comments.

Be prepared for the next section where we will have a working parallel program.

Chapter 2. A working parallel program

Now we want to gather the results of the computation in one place and eventually write them out to a file. My first try to do that looked like `apple_mpi.1.c` (Appendix G).

The root process sets up an MPI_Irecv for each row and uses the TAG to let each row result fall into place. This will work as long as the upper bound for TAG values will be bigger than the number of rows.

All ranks send the calculated rows back to the root, even root itself. I assume that each serious MPI implementation will recognize this particular case and will reduce the MPI_Send to a memcpy. This makes the code very symmetric on all ranks.

Now what do we get when we run that with two CPUs?

```
> time mpiexec -np 1 ./apple_mpi.1
...
real    0m5.161s
user    0m0.120s
sys     0m0.012s
```

That's ok. How about 2 processes?

```
> time mpiexec -np 2 ./apple_mpi.1
...
real    0m4.887s
user    0m0.104s
sys     0m0.024s
```

Now that is a real mess! We invested all this effort and on two CPUs we get a performance that is equal to the serial program using only one CPU. Ok, lets forget about this MPI thing and buy a faster single CPU machine then...

Well, this kind of disappointment will stay with us. We just have to find out what happened. We used the MPI_Irecv to give the MPI a chance to receive the messages behind the scenes while we are doing computational work. Good idea. But apparently it didn't work.

You can do performance analysis the hard way using print statements in your code - if you can afford this waste of time. I suggest to start with a evaluation copy of the Intel Trace Analyzer and Collector (ITAC) to do performance analysis and correctness checking of your MPI programs.

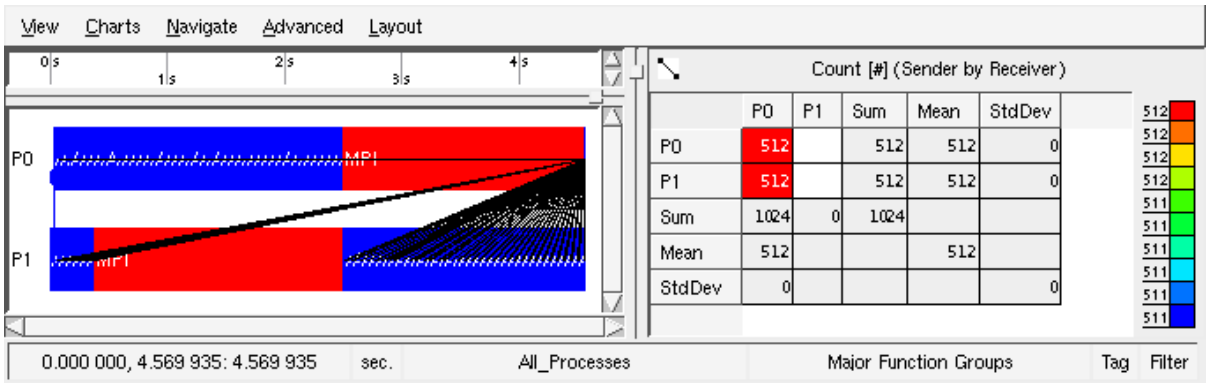
If ITAC is properly installed and the binary is linked against shared MPI libraries then getting performance data for our program can be as simple as this:

```
> LD_PRELOAD=libVT.so time mpiexec -np 2 ./apple_mpi.1
...
[0] Intel(R) Trace Collector INFO: Writing tracefile apple_mpi.1.stf in /home/georg/src/
0.11user 0.02system 0:05.66elapsed 2%CPU (0avgtext+0avgdata 0maxresident)k
```

```
0inputs+0outputs (0major+2526minor)pagefaults 0swaps
```

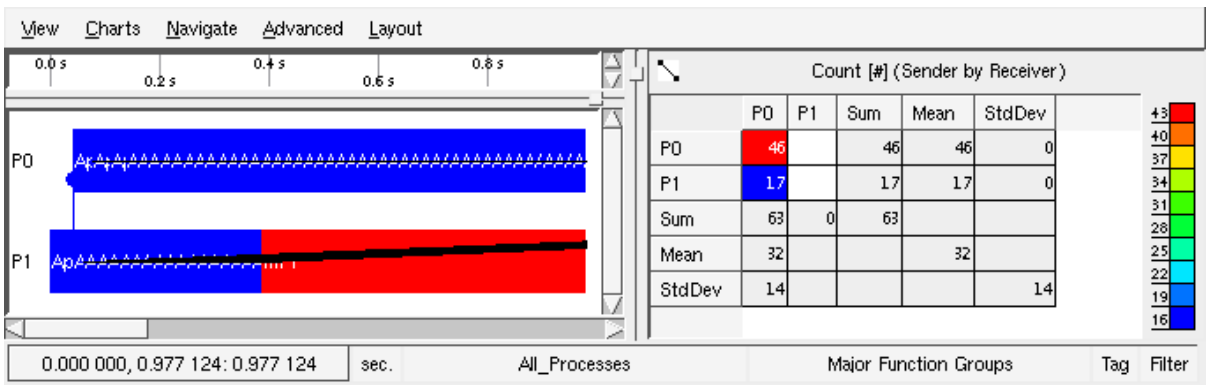
ITAC (Figure 2-1) shows that the first process P0 goes straight into the calculation while P1 calculates some rows, sends them back and get's stuck. Only after P0 finished it's calculations P1 can complete it's task. The diagram to the right shows that both processes sent 512 rows back to P0, as expected.

Figure 2-1. The whole mess.



Zooming into the first second shows that P1 is able to send 16 rows back before the 17th MPI_Send blocks (Figure 2-2).

Figure 2-2. The first second of the mess.



If you refer to the MPI standard you see that the MPI implementation is well within the specs. So we have to change our tactics. We will now for the first time use the `IDLEROOT` feature. Note that we use three processors, one for the idle root and two to actually calculate:

```
>time mpiexec -np 3 ./apple_mpi.1.ir
...
real    0m2.839s
user    0m0.100s
sys     0m0.016s
```

That looks reasonable. But we waste a whole processor for the idle root process. It can be a very good thing to reserve an extra processor for bookkeeping tasks or for doing things that have to be serialized (I/O libraries for example tend to lack parallelism).

What if we would interrupt the calculation of P0 after each row to poll for the messages coming in from the other processes? The program `apple_mpi_poll.1.c` (Appendix H) tries to do that:

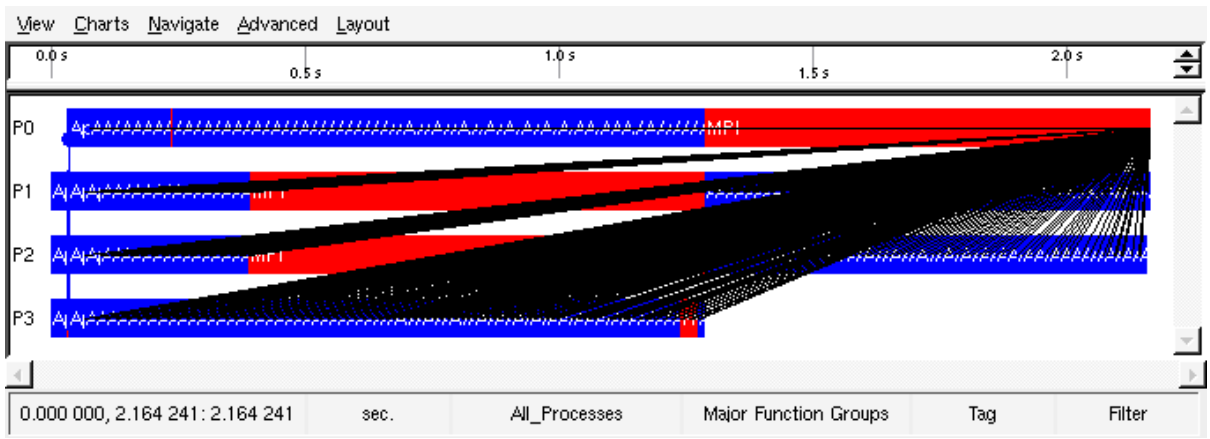
```
> time mpiexec -np 2 ./apple_mpi_poll.1
...
real    0m2.846s
user    0m0.108s
sys     0m0.020s
```

That look's ok. How about using all four processors?

```
> time mpiexec -np 4 ./apple_mpi_poll.1
...
real    0m2.525s
user    0m0.092s
sys     0m0.020s
```

Ouch! What happens here? Again we use ITAC for analysis (Figure 2-3) and see that the polling does not really allow P1 and P2 to make progress.

Figure 2-3. Polling does not help.



We see that this program does not behave much better than apple_mpi.1.c. For some reason the polling does not work as we expect. In the next chapter we'll try to do better by using MPI's non-blocking point to point communication.

Chapter 3. Entirely nonblocking

We will now not only use MPI_Irecv at rank 0 to collect the data but we will use MPI_Isend during the calculation. Our program does fit this strategy very well because it uses another piece of memory for each result so that we do not have to allocate extra buffers for the use of MPI_Isend. The result of this effort is `apple_mpi_nblock.1.c` (Appendix I). Let's see:

```
> time mpirun -np 2 ./apple_mpi_nblock.1
...
real    0m2.843s
user    0m0.100s
sys     0m0.020s
> time mpirun -np 4 ./apple_mpi_nblock.1
...
real    0m1.663s
user    0m0.104s
sys     0m0.008s
```

That looks fine. But this program behaves really bad with an MPI implementation from the mpich-1.2.x branch. If this nonblocking program will not run acceptable on your MPI then try `apple_mpi_nblock.1.ir` (Appendix I) or `apple_mpi_nblock.2.c` (Appendix J).

In the next section we will switch from static assignment to a dynamic scheme.

Chapter 4. Dynamic load balancing

Why do we need dynamic load balancing? We have several potential causes that could result in varying run time on our processes. The most apparent reason is that the computers we have may differ in their processing power. Another reason is that different areas of the Mandelbrot set can require very different amounts of computation. In this particular case the scheme of interleaving the rows onto processors works quite well, but what if such a simple scheme is not known beforehand? And there could of course be varying load on the machines caused by those other users that unfortunately share the computers with us or by some demon processes that we forgot to switch off.

Generally speaking there is often no way to find an adequate scheduling of the work to the available processors at startup time. So we should come up with a scheme that splits the work into packets and assigns them to the next free process. This way faster CPUs will be able to do more work than slow ones. There is a tradoff here: small packets cost bandwidth, but they allow for a fine grained adaption of the work load to the processors.

The program `apple_mpi_dynamic.1.c` (Appendix H) tries to do that. Note that it is only available in the `IDLEROOT` flavor. But since the root process will not have that much to do we dare to start 5 processes on our 4 core box:

```
> time mpiexec -np 5 ./apple_mpi_dynamic.1.ir
...
real    0m1.830s
user    0m0.100s
sys     0m0.024s
```

Works like a charm, doesn't it? The overhead compared to the simple non-blocking programs is quite high but keep in mind that we get a program that will adapt to load changes on the processors and imbalances in the task.

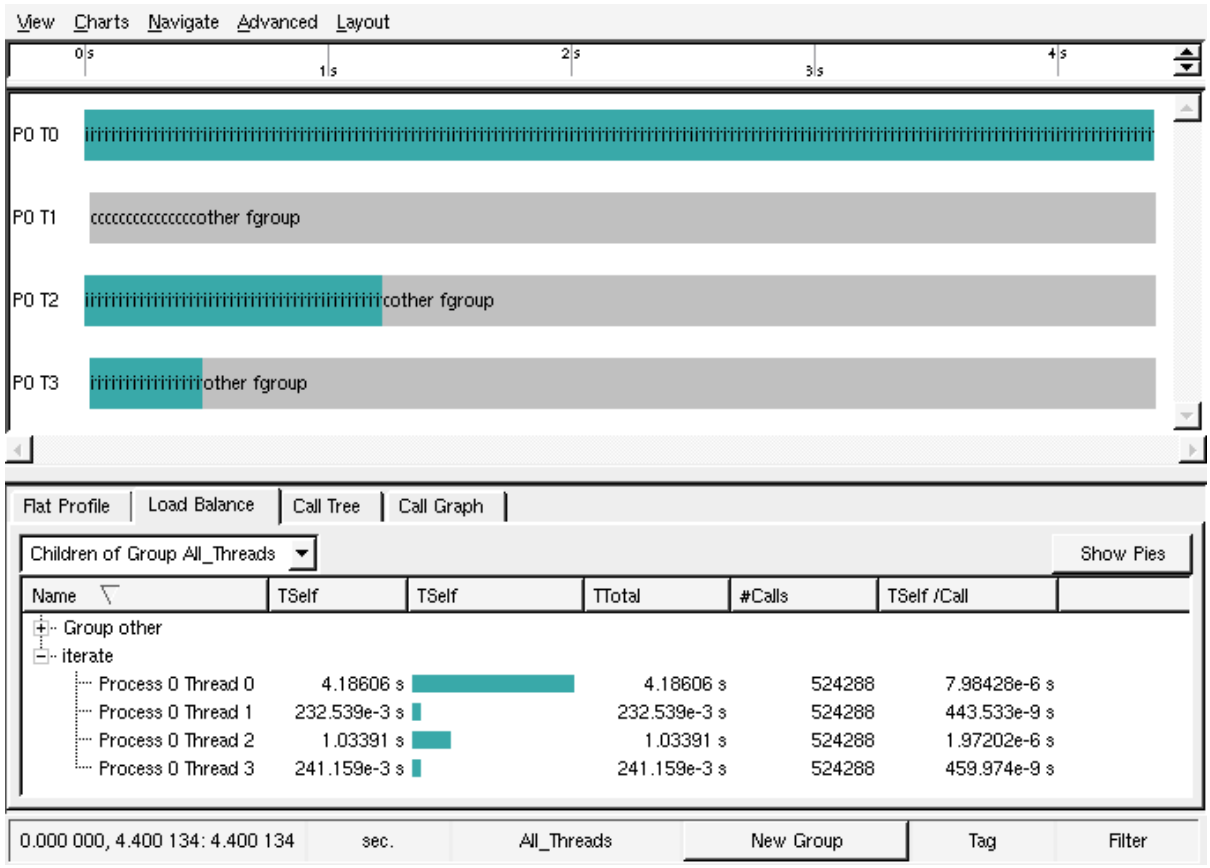
Chapter 5. Using OpenMP

With the advance of relatively cheap multi-core CPU's it has become much more attractive to use OpenMP. In contrast to MPI it will only work on SMP boxes and not on clusters of networked computing nodes, but at first sight OpenMP's programming model seems so much simpler than MPI's that it seems worth a try. As Oscar Wilde said: "I can resist anything - but temptation.".

My first try to use OpenMP looked similar to `apple_omp.1.c` (Appendix L). Unfortunately this does not work too good:

```
> time ./apple_omp.1
...
real    0m4.078s
user    0m5.424s
sys     0m0.016s
```

Figure 5-1. OpenMP with initial loadbalancing problems.



This program ran on 4 cores using 4 threads and was not much better than the serial version. This is a little disappointing. The reason is that in apple_omp.1 a very dumb static scheduling is used. The

range of rows is just cut into four areas so that one thread gets all the expensive rows that belong to the Mandelbrodt set. Refer to Figure 5-1: the computing function iterate is colored while everything else is gray. The Load Balance tab of the function profile shows that one thread does the lions share of the computation.

This tracefile for this non-MPI program was generated by using the trace collectors API and then linking against a null implementation (see the Makefile). Running this binary yields next to no overhead but no tracefile either. Using binary instrumentation (man itcinstrument) a fully instrumented binary was produced.

Adding a single directive **`schedule(dynamic, 1)`** to the OpenMP pragma results in `apple_omp.2.c` (Appendix M) and solves the problem :

```
> time ./apple_omp.2
...
real    0m1.263s
user    0m4.916s
sys     0m0.016s
```

Now that is fast! For this particular program and on this small SMP box OpenMP seems to be the tool of choice. The advantage of OpenMP is that it seems so simple to parallelize an application without doing any restructuring of the code base. But after harvesting some low hanging fruits you might end up with the conclusion that further performance improvements can only be done with a restructuring of the code.

For those programers who still are hoping for a free lunch when parallelizing applications and porting them to clusters there is still hope: Cluster OmpenMP or clomp for short (sounds like the wooden shoes of the Dutch). This is available from Intel as an additional product on top of their C++ and Fortran compilers for x86_64-Linux.

Despite the fact that clomp is a lot more lightweight than creating a single system image illusion in a cluster it can fail to deliver acceptable performance when an application shares to much data for read and write in a parallel region. As the rumours have it, there are some OpenMP codes that scale surprisingly well with clomp. I never saw that myself, but I guess apple_omp.2 would be a good candidate.

Appendix A. invar.h

```
#ifndef _INVAR_H_
#define _INVAR_H_

/* Please see invar.c for a comment. */

void usage (int argc, char **argv);
int initRanges (int argc, char **argv);
void distributeRanges (void);
int iterate (double re, double im, int maxiter);
void writeOut (char *progname, int nprocs, int *result);

#ifndef INVAR_EXTERN
#define INVAR_EXTERN extern
#endif

INVAR_EXTERN double RE_START;
INVAR_EXTERN double RE_STOP;
INVAR_EXTERN double IM_START;
INVAR_EXTERN double IM_STOP;

INVAR_EXTERN int XPIX;
INVAR_EXTERN int YPIX;
INVAR_EXTERN int MAXITER;

#undef INVAR_EXTERN

/*
   These are needed in mpiapple0.1 and above to distribute
   the rows to ranks. Notice that we can decide if rank 0
   should participate in the calculation or not. This is
   used to implement a master/slave scheme.
*/
#ifdef IDLEROOT
#define MYROW(_nprocs, _myrank, _row) (_myrank - 1 == _row % (_nprocs - 1))
#define PROCLIMIT 2
#else
#define MYROW(_nprocs, _myrank, _row) (_myrank == _row % _nprocs)
#define PROCLIMIT 1
#endif


#endif
```

Appendix B. invar.c

```
/* define global vars: */
#define INVAR_EXTERN  /* */

#include "invar.h"
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <libgen.h>

/* Tell what command line arguiments we understand. */
void
usage (int argc, char **argv)
{
    char *name = "<progname>";
    if (argc>0) name = argv[0];
    fprintf (stderr,
      "Usage is either:\n"
      "%s XPIX RE_START RE_STOP YPIX IM_START IM_STOP MAXITER\n"
      "to set everything.\n\n", name);
  fprintf (stderr, "Or:\n" "%s <n>\n" "to set only MAXITER=<n>\n\n", name);
  fprintf (stderr,
    "Or simply:\n"
    "%s\n" "to leave everything at its default.\n", name);
}

/* Init the rectangle in the complex plane, the resolution and
   the maximum number of iterations per pixel.
*/
int
initRanges (int argc, char **argv)
{
  int keep_going = 1;  // ==1 mwans parsing ok

  RE_START = -2.0;
  IM_START = 0.0;
  RE_STOP = IM_STOP = +2.0;
  XPIX = 2048;
  YPIX = 1024;
  MAXITER = 4096;

  if (8 == argc) {
      if (1 != sscanf (argv[1], "%d", &XPIX))
 keep_going = 0;
      if (1 != sscanf (argv[2], "%lg", &RE_START))
 keep_going = 0;
      if (1 != sscanf (argv[3], "%lg", &RE_STOP))
 keep_going = 0;
      if (1 != sscanf (argv[4], "%d", &YPIX))
 keep_going = 0;
      if (1 != sscanf (argv[5], "%lg", &IM_START))
 keep_going = 0;
      if (1 != sscanf (argv[6], "%lg", &IM_STOP))
```

```
 keep_going = 0;
      if (1 != sscanf (argv[7], "%d", &MAXITER))
 keep_going = 0;
  } else if (2 == argc) {
      if (1 != sscanf (argv[1], "%d", &MAXITER))
 keep_going = 0;
  } else if (1 == argc) {
      ;
  } else {
      keep_going = 0;
  }

  if (keep_going) {
      fprintf(stdout, "RE_START = %40.30g\n", RE_START);
      fprintf(stdout, "RE_STOP  = %40.30g\n", RE_STOP);
      fprintf(stdout, "IM_START = %40.30g\n", IM_START);
      fprintf(stdout, "IM_STOP  = %40.30g\n", IM_STOP);
      fprintf(stdout, "XPIX     = %40d\n",    XPIX);
      fprintf(stdout, "YPIX     = %40d\n",    YPIX);
      fprintf(stdout, "MAXITER  = %40d\n",    MAXITER);
      fflush (stdout);
    }

  return keep_going;
}

void
distributeRanges (void)
{
  MPI_Bcast(&XPIX,     1, MPI_INT,    0, MPI_COMM_WORLD);
  MPI_Bcast(&RE_START, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&RE_STOP,  1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&YPIX,     1, MPI_INT,    0, MPI_COMM_WORLD);
  MPI_Bcast(&IM_START, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&IM_STOP,  1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&MAXITER,  1, MPI_INT,    0, MPI_COMM_WORLD);
}

int
iterate (double re, double im, int maxiter)
{
  double x = 0, y = 0, x2 = 0, y2 = 0;
  int k = 0;

  while ((k < maxiter) && (x2 + y2 <= ((double) 4.0)))
    {
      y = ((double) 2.0) * x * y + im;
      x = x2 - y2 + re;
      x2 = x * x;
      y2 = y * y;
      k++;
    }

  return k;
}

/* write it out as a 16 bit grayscale image suitable
```

```
   to be displayed with ImageMagick like this:
   display -depth 16 -size XPIYxYPIX <name>
   Use Enhance->Equalize to see something.
   This is not to really look terribly nice
   but only to have a way to control that the
   calculated results look reasonable.
*/
void
writeOut (char *progname, int nprocs, int *result)
{
  char fname[256] = { 0 };
  FILE *fp;
  int row, col;
  unsigned int truncated;
  unsigned char clow, chigh;

  if (0 == getenv ("WRITEOUT"))
    return;

  sprintf(fname,
   "%s_%dp_size%dx%d.gray",
   basename(progname), nprocs, XPIX, YPIX);
  fprintf (stdout, "Writing result to file \"%s\".\n", fname);

  fp = fopen (fname, "w");
  for (row = YPIX - 1; row >= 0; row--)
      for (col = 0; col < XPIX; col++) {
   /* The "% MAXITER" makes the inside of the apple black
      as I've gotten used to that. At least for non eatable
      apples. */
   truncated = ((result[row * XPIX + col] % MAXITER) % 0x10000);
   clow  = (unsigned char) (truncated & 0xff);
   chigh = (unsigned char) ((truncated >> 8) & 0xff);
   fwrite(&chigh, 1, 1, fp);
   fwrite(&clow,  1, 1, fp);
      }
  fclose (fp);
}
```

Appendix C. apple_serial.c

```
/*
  DESC: apple_serial.c is a simple sequential program to start with.
*/

#include <stdio.h>
#include <stdlib.h>
#include "invar.h"

int main (int argc, char **argv)
{
  int  row, col, iter, keep_going;
  int   *result = 0;
  double  re, im;

  keep_going = initRanges (argc, argv);
  if (!keep_going) {
      usage (argc, argv);
      exit (-1);
  }

  result = (int *) malloc (XPIX * YPIX * sizeof (*result));

  for (row = 0; row < YPIX; row++) {
      im = IM_START + row * (IM_STOP - IM_START) / (YPIX - 1.0);
      for (col = 0; col < XPIX; col++) {
   re = RE_START + col * (RE_STOP - RE_START) / (XPIX - 1.0);
   iter = iterate (re, im, MAXITER);
   result[row * XPIX + col] = iter;
      }
      //fprintf (stdout, "%8d\r", row); fflush (stdout);
  }

  writeOut (argv[0], 1, result);
  return 0;
}
```

Appendix D. apple_pseudo_mpi.1.c

```
/*
  DESC: using MPI only for its overhead.
  Just demonstrate the overhead in initialization and
  finalization that MPI introduces.
  This program makes no intelligent use of more than
  one processor: if you run it on n>=1 CPUs then it
  will do the whole work on each CPU.
*/

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "invar.h"

int main (int argc, char **argv)
{
  int   row, col, iter, keep_going, nprocs, myrank;
  int   *result = 0;
  double  re, im;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);

  keep_going = initRanges (argc, argv);
  if (!keep_going) {
      usage (argc, argv);
      exit (-1);
  }

  result = (int *) malloc (XPIX * YPIX * sizeof (*result));

  for (row = 0; row < YPIX; row++) {
      im = IM_START + row * (IM_STOP - IM_START) / (YPIX - 1.0);
      for (col = 0; col < XPIX; col++) {
   re = RE_START + col * (RE_STOP - RE_START) / (XPIX - 1.0);
   iter = iterate (re, im, MAXITER);
   result[row * XPIX + col] = iter;
      }
      fprintf (stdout, "%8d\r", row);
      fflush (stdout);
  }
  fprintf (stdout, "\n");

  MPI_Finalize ();

  writeOut (argv[0], nprocs, result);
  return 0;
}
```

Appendix E. apple_pseudo_mpi.2.c

```
/*
  DESC: calculating parallel and getting no results.
  We start with the parallelization of the calculation,
  by assigning each row to a different processor.
  For now we do not solve the problem of gathering
  the results.
*/

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "invar.h"

int main (int argc, char **argv)
{
  int   row, col, iter, keep_going, nprocs, myrank;
  int   *result = 0;
  double  re, im;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);

  if (nprocs < PROCLIMIT) {
      MPI_Finalize ();
      exit (-1);
  }

  keep_going = initRanges (argc, argv);
  if (!keep_going) {
      usage (argc, argv);
      exit (-1);
  }

  result = (int *) malloc (XPIX * YPIX * sizeof (*result));

#ifdef IDLEROOT
  if (myrank != 0)
#endif
    for (row = 0; row < YPIX; row++)
      if (MYROW (nprocs, myrank, row)) {
   im = IM_START + row * (IM_STOP - IM_START) / (YPIX - 1.0);
   for (col = 0; col < XPIX; col++) {
       re = RE_START + col * (RE_STOP - RE_START) / (XPIX - 1.0);
       iter = iterate (re, im, MAXITER);
       result[row * XPIX + col] = iter;
     }
 }

  MPI_Finalize ();

  if (0 == myrank) writeOut (argv[0], nprocs, result);
  return 0;
```

```
}
```

Appendix F. apple_pseudo_mpi.3.c

```
/*
  DESC: calculating parallel and getting no results, continued.
  This is just a small improvement over mpiapple0.1.
  Only rank 0 will evaluate the command line arguments
  and we will use MPI_Bcast to pass the result of
  this evaluation to the other ranks. If the evaluation fails
  all ranks will terminate using MPI_Finalize in an orderly
  fashion.
  We still do not gather the results in one place.
*/

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "invar.h"

int main (int argc, char **argv)
{
  int   row, col, iter, keep_going, nprocs, myrank;
  int   *result = 0;
  double  re, im;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);

  if (nprocs < PROCLIMIT) {
      MPI_Finalize ();
      exit (-1);
  }

  if (0 == myrank) keep_going = initRanges (argc, argv);
  MPI_Bcast (&keep_going, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (!keep_going) {
      if (0 == myrank) usage (argc, argv);
      MPI_Finalize ();
      exit (-1);
  }

  distributeRanges ();

  result = (int *) malloc (XPIX * YPIX * sizeof (*result));

#ifdef IDLEROOT
  if (myrank != 0)
#endif
    for (row = 0; row < YPIX; row++)
      if (MYROW (nprocs, myrank, row)) {
   im = IM_START + row * (IM_STOP - IM_START) / (YPIX - 1.0);
   for (col = 0; col < XPIX; col++) {
       re = RE_START + col * (RE_STOP - RE_START) / (XPIX - 1.0);
       iter = iterate (re, im, MAXITER);
       result[row * XPIX + col] = iter;
```

```
    }
       }

  MPI_Finalize ();

  if (0 == myrank) writeOut (argv[0], nprocs, result);
  return 0;
}
```

Appendix G. apple_mpi.1.c

```
/*
  DESC: the first working MPI program.
  Despite the complexity the performance is not too rewarding.
  Reason is the (potentially) blocking MPI_Send that is used
  to report the rows back.
*/

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "invar.h"

int main (int argc, char **argv)
{
  int   row, col, iter, keep_going, nprocs, myrank;
  int   *result = 0;
  double  re, im;
  MPI_Request  *recv_reqs = 0;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);

  if (nprocs < PROCLIMIT) {
      MPI_Finalize ();
      exit (-1);
  }

  if (0 == myrank) keep_going = initRanges (argc, argv);
  MPI_Bcast (&keep_going, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (!keep_going) {
      if (0 == myrank)usage (argc, argv);
      MPI_Finalize ();
      exit (-1);
  }

  distributeRanges ();

  /* for simplicity we allow the full matrix in each rank: */
  result = (int *) malloc (XPIX * YPIX * sizeof (*result));

  /* set up the MPI_Irecvs needed. We use the tag of the
     MPI messages to let each result fall into place:
   */
  if (0 == myrank) {
      recv_reqs = (MPI_Request *) malloc (YPIX * sizeof (MPI_Request));
      for (row = 0; row < YPIX; row++)
   MPI_Irecv(result + (row * XPIX), XPIX, MPI_INT,
      MPI_ANY_SOURCE /* we do not care who does the job */ ,
      row /* tag */ ,
      MPI_COMM_WORLD, recv_reqs + row);
  }
```

```
#ifdef IDLEROOT
  if (myrank != 0)
#endif
    for (row = 0; row < YPIX; row++)
      if (MYROW (nprocs, myrank, row)) {
   im = IM_START + row * (IM_STOP - IM_START) / (YPIX - 1.0);
   for (col = 0; col < XPIX; col++) {
       re = RE_START + col * (RE_STOP - RE_START) / (XPIX - 1.0);
       iter = iterate (re, im, MAXITER);
       result[row * XPIX + col] = iter;
   }
   /* report row back to root: */
   MPI_Send(result + row * XPIX, XPIX, MPI_INT,
     0, row, MPI_COMM_WORLD);
      }

  /* complete all Irecvs: */
  if (0 == myrank) MPI_Waitall (YPIX, recv_reqs, MPI_STATUSES_IGNORE);

  MPI_Finalize ();

  if (0 == myrank) writeOut (argv[0], nprocs, result);
  return 0;
}
```

Appendix H. apple_mpi_poll.1.c

```
/*
  DESC: rank 0 polls for results, other ranks send blocking
  This tries to be a little better than apple_mpi.1.c by
  letting rank 0 poll for results during it's own
  calculation. This should prevent the other ranks from
  beeing blocked.
*/

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "invar.h"

int main (int argc, char **argv)
{
  int   row, col, iter, keep_going, nprocs, myrank;
  int   *result = 0;
  double  re, im;
  MPI_Request  *recv_reqs = 0;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);

  if (nprocs < PROCLIMIT) {
      MPI_Finalize ();
      exit (-1);
  }

  if (0 == myrank) keep_going = initRanges (argc, argv);
  MPI_Bcast (&keep_going, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (!keep_going) {
      if (0 == myrank) usage (argc, argv);
      MPI_Finalize ();
      exit (-1);
  }

  distributeRanges ();

  result = (int *) malloc (XPIX * YPIX * sizeof (*result));

  if (0 == myrank) {
      recv_reqs = (MPI_Request *) malloc (YPIX * sizeof (MPI_Request));
      for (row = 0; row < YPIX; row++)
   MPI_Irecv(result + (row * XPIX), XPIX, MPI_INT,
      MPI_ANY_SOURCE,
      row,
      MPI_COMM_WORLD, recv_reqs + row);
  }

#ifdef IDLEROOT
  if (myrank != 0)
#endif
```

```
    for (row = 0; row < YPIX; row++)
      if (MYROW(nprocs, myrank, row)) {
   im = IM_START + row * (IM_STOP - IM_START) / (YPIX - 1.0);
   for (col = 0; col < XPIX; col++) {
       re = RE_START + col * (RE_STOP - RE_START) / (XPIX - 1.0);
       iter = iterate (re, im, MAXITER);
       result[row * XPIX + col] = iter;
   }
   MPI_Send(result + row * XPIX, XPIX, MPI_INT,
     0, row, MPI_COMM_WORLD);
   /* If root calculates it polls for results during calculation: */
#ifndef IDLEROOT
   if (0 == myrank) {
       int flag;
       /* Does it make a difference? */
#if 0
       int index;
       MPI_Testany(YPIX, recv_reqs, &index, &flag, MPI_STATUSES_IGNORE);
#else
       MPI_Testall(YPIX, recv_reqs,         &flag, MPI_STATUSES_IGNORE);
#endif
     }
#endif
      }

  if (0 == myrank) MPI_Waitall (YPIX, recv_reqs, MPI_STATUSES_IGNORE);

  MPI_Finalize ();

  if (0 == myrank) writeOut (argv[0], nprocs, result);
  return 0;
}
```

Appendix I. apple_mpi_nblock.1.c

```
/*
  DESC: rank 0 polls, other ranks send nonblocking - at least they try.
  This tries to be a little better than apple_mpi_poll.1.c by using
  MPI_Isend to send back the intermediate results to rank 0.
  Hint: Depending on your MPI implementation and device/interconnect
        that might not work as expected.
*/

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "invar.h"

int main (int argc, char **argv)
{
  int   row, col, iter, keep_going, nprocs, myrank,
  numreqs = 0;
  int   *result = 0;
  double  re, im;
  MPI_Request  *reqs = 0; /* recv for rank 0, send for everybody else */

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);

  if (nprocs < PROCLIMIT) {
      MPI_Finalize ();
      exit (-1);
  }

  if (0 == myrank) keep_going = initRanges (argc, argv);
  MPI_Bcast (&keep_going, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (!keep_going) {
      if (0 == myrank) usage (argc, argv);
      MPI_Finalize ();
      exit (-1);
  }

  distributeRanges ();

  result = (int *) malloc (XPIX * YPIX * sizeof (*result));

  /* wastefully allocate memory for requests */
  reqs = (MPI_Request *) malloc (2 * YPIX * sizeof (MPI_Request));

  if (0 == myrank) {
      for (row = 0; row < YPIX; row++) {
   MPI_Irecv(result + (row * XPIX), XPIX, MPI_INT,
      MPI_ANY_SOURCE,
      row,
      MPI_COMM_WORLD, reqs + numreqs);
   numreqs++;
      }
```

```
  }

#ifdef IDLEROOT
  if (myrank != 0)
#endif
    for (row = 0; row < YPIX; row++)
      if (MYROW (nprocs, myrank, row)) {
   im = IM_START + row * (IM_STOP - IM_START) / (YPIX - 1.0);
   for (col = 0; col < XPIX; col++) {
       re = RE_START + col * (RE_STOP - RE_START) / (XPIX - 1.0);
       iter = iterate (re, im, MAXITER);
       result[row * XPIX + col] = iter;
   }
   MPI_Isend (result + row * XPIX, XPIX, MPI_INT,
       0, row, MPI_COMM_WORLD, reqs + numreqs);
   numreqs++;
 }

  /* complete all nonblocking communication: */
  MPI_Waitall (numreqs, reqs, MPI_STATUSES_IGNORE);

  MPI_Finalize ();

  if (0 == myrank) writeOut (argv[0], nprocs, result);
  return 0;
}
```

Appendix J. apple_mpi_nblock.2.c

```
/*
  DESC: first calculate, then communicate.
  To circumvent the problems of apple_mpi_nblock.1.c we delay the
  communication. No viable option for real world problems...
*/

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "invar.h"

int main (int argc, char **argv)
{
  int   row, col, iter, keep_going, nprocs, myrank,
  numreqs = 0;
  int   *result = 0;
  double  re, im;
  MPI_Request  *reqs = 0; /* recv for rank 0, send for everybody else */

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);

  if (nprocs < PROCLIMIT) {
      MPI_Finalize ();
      exit (-1);
  }

  if (0 == myrank) keep_going = initRanges (argc, argv);
  MPI_Bcast (&keep_going, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (!keep_going) {
      if (0 == myrank) usage (argc, argv);
      MPI_Finalize ();
      exit (-1);
  }

  distributeRanges ();

  result = (int *) malloc (XPIX * YPIX * sizeof (*result));

  reqs = (MPI_Request *) malloc (2 * YPIX * sizeof (MPI_Request));

  if (0 == myrank) {
      for (row = 0; row < YPIX; row++) {
   MPI_Irecv (result + (row * XPIX), XPIX, MPI_INT,
       MPI_ANY_SOURCE,
       row,
       MPI_COMM_WORLD, reqs + numreqs);
   numreqs++;
      }
  }

#ifdef IDLEROOT
```

```
  if (myrank != 0)
#endif
    for (row = 0; row < YPIX; row++)
      if (MYROW (nprocs, myrank, row)) {
   im = IM_START + row * (IM_STOP - IM_START) / (YPIX - 1.0);
   for (col = 0; col < XPIX; col++) {
       re = RE_START + col * (RE_STOP - RE_START) / (XPIX - 1.0);
       iter = iterate (re, im, MAXITER);
       result[row * XPIX + col] = iter;
   }
   /* no communication here! */
      }

  /* now let everybody send results back, even root if necessary */
#ifdef IDLEROOT
  if (myrank != 0)
#endif
      for (row = 0; row < YPIX; row++)
   if (MYROW (nprocs, myrank, row)) {
       MPI_Isend (result + row * XPIX, XPIX, MPI_INT,
    0, row, MPI_COMM_WORLD, reqs + numreqs);
       numreqs++;
   }

  MPI_Waitall (numreqs, reqs, MPI_STATUSES_IGNORE);

  MPI_Finalize ();

  if (0 == myrank) writeOut (argv[0], nprocs, result);
  return 0;
}
```

Appendix K. apple_mpi_dynamic.1.c

```
/*
  DESC: dynamic load balancing
  This demonstrates how to adapt to varying calculation times due
  to imbalance in work packets (rows) or due to varying or
  asymetric computing power in the processors.
  This uses a master-slaves scheme.
 */

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "invar.h"

#define TAG_WORK 0x0815 /* flags an order to a slave */
#define TAG_WORK_DONE 0x1704 /* flags a result from a slave */

int main (int argc, char **argv)
{
  int   col, iter, keep_going = 1, nprocs, myrank, rank;
  int   *result = 0;
  int   *lcount = 0; /* lcount[p] == #rows processor p did */
  double  re, im;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);

  if (nprocs < PROCLIMIT) {
      MPI_Finalize ();
      exit (-1);
  }

  if (0 == myrank) keep_going = initRanges (argc, argv);
  MPI_Bcast (&keep_going, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (!keep_going) {
      if (0 == myrank)usage (argc, argv);
      MPI_Finalize ();
      exit (-1);
  }

  distributeRanges ();

  if (0 == myrank) {
      /* master */
      int i;
      int first_run = 1;
      int numbusy = 0, nextline = YPIX - 1;
      int *busy = 0; /* busy[p] != 0 means processor p is busy */
      MPI_Request *rreq = 0;

      lcount = (int *) malloc (nprocs * sizeof (*lcount));
      memset (lcount, 0, nprocs * sizeof (*lcount));
```

```
    result = (int *) malloc (XPIX * YPIX * sizeof (*result));

    /* allocate mem in block local vars */
    busy = (int *) malloc (nprocs * sizeof (*busy));
    memset (busy, 0, nprocs * sizeof (*busy));

    rreq = (MPI_Request *) malloc (nprocs * sizeof (*rreq));
    /* you can not do that standard compliant with memset() */
    for (i = 0; i < nprocs; i++) rreq[i] = MPI_REQUEST_NULL;

    /* loop while there is work to do or some ranks are still working */
    while (nextline >= 0 || numbusy > 0) {
  /* send out work */
  if (nextline >= 0 && numbusy < nprocs - 1) {
      for (rank = 1; rank < nprocs; rank++)
   if (0 == busy[rank]) {
       busy[rank] = 1;
       /* slave listens, so blocking send is ok */
       MPI_Send(&nextline, 1, MPI_INT, rank, TAG_WORK,
   MPI_COMM_WORLD);
       MPI_Irecv(result + nextline * XPIX, XPIX, MPI_INT,
   rank, TAG_WORK_DONE, MPI_COMM_WORLD,
   rreq + rank);
       nextline--;
       numbusy++;
       /* make all busy in the first run,
   otherwise only one per loop */
       if (!first_run)
    break;
   }
  }
  first_run = 0;
  /* try to receive something */
  if (numbusy > 0) {
      int who;
      MPI_Waitany (nprocs, rreq, &who, MPI_STATUS_IGNORE);
      if (MPI_UNDEFINED != who) {
   busy[who] = 0;
   lcount[who]++;
   numbusy--;
      }
  }
    }

    /* send all slaves the signal to stop: */
    for (rank = 1; rank < nprocs; rank++)
  MPI_Send (&nextline, 1, MPI_INT, rank, TAG_WORK, MPI_COMM_WORLD);

} else {
    /* slave(s) */
    int row;

    result = (int *) malloc (XPIX * sizeof (*result));

    while (keep_going) {
  MPI_Recv (&row, 1, MPI_INT, 0, TAG_WORK, MPI_COMM_WORLD,
```

```
      MPI_STATUS_IGNORE);
   if (-1 == row) {
       keep_going = 0;
   } else {
       im = IM_START + (row * (IM_STOP - IM_START)) / (YPIX - 1.0);
       for (col = 0; col < XPIX; col++) {
    re = RE_START + (col * (RE_STOP - RE_START)) / (XPIX - 1.0);
    iter = iterate (re, im, MAXITER);
    result[col] = iter;
       }
       MPI_Send(result, XPIX, MPI_INT,
         0, TAG_WORK_DONE, MPI_COMM_WORLD);
   }
      }
  }

  if (0 == myrank) {
      for(int r=0; r<nprocs; r++)
   fprintf(stdout, "%4d: calculated %5d lines\n", r, lcount[r]);
  }

  fflush(stdout);

  MPI_Finalize ();

  if (0 == myrank) writeOut (argv[0], nprocs, result);
  return 0;
}
```

Appendix L. apple_omp.1.c

```
#include <stdio.h>
#include <stdlib.h>
#ifdef USEVT
#include <VT.h>
#endif
#include "invar.h"
int
main (int argc, char **argv)
{
    int  row, col, iter, keep_going;
    double  re, im;
#ifdef COMPARE
    int  *result_serial = 0;
#endif
    int  *result_parallel = 0;

#ifdef USEVT
    VT_initialize(&argc, &argv);
#endif

    keep_going = initRanges (argc, argv);
    if (!keep_going) {
 usage (argc, argv);
 exit (-1);
    }

#ifdef COMPARE
    result_serial   = (int*) malloc(XPIX * YPIX * sizeof (*result_serial));
    for (row = 0; row < YPIX; row++) {
 im = IM_START + (row * (IM_STOP - IM_START)) / (YPIX - 1.0);
 for (col = 0; col < XPIX; col++) {
     re = RE_START + (col * (RE_STOP - RE_START)) / (XPIX - 1.0);
     iter = iterate(re, im, MAXITER);
     result_serial[row * XPIX + col] = iter;
 }
    }
#endif

    result_parallel = (int*) malloc(XPIX * YPIX * sizeof (*result_parallel));
#   pragma omp parallel for \
               private(row, col, re, im, iter) \
               shared(result_parallel)
    for (row = 0; row < YPIX; row++) {
 im = IM_START + (row * (IM_STOP - IM_START)) / (YPIX - 1.0);
 for (col = 0; col < XPIX; col++) {
     re = RE_START + (col * (RE_STOP - RE_START)) / (XPIX - 1.0);
     iter = iterate(re, im, MAXITER);
     result_parallel[row * XPIX + col] = iter;
 }
    }
#ifdef COMPARE
    for (row = 0; row < YPIX; row++) {
 for (col = 0; col < XPIX; col++) {
```

```
     if (result_serial[row * XPIX + col]
  != result_parallel[row * XPIX + col] )
  return -1;
 }
    }
#endif

#ifdef USEVT
    VT_finalize();
#endif

    return 0;
}
```

Appendix M. apple_omp.2.c

```
#include <stdio.h>
#include <stdlib.h>
#ifdef USEVT
#include <VT.h>
#endif
#include "invar.h"
int
main (int argc, char **argv)
{
    int  row, col, iter, keep_going;
    double  re, im;
#ifdef COMPARE
    int  *result_serial = 0;
#endif
    int  *result_parallel = 0;

#ifdef USEVT
    VT_initialize(&argc, &argv);
#endif

    keep_going = initRanges (argc, argv);
    if (!keep_going) {
 usage (argc, argv);
 exit (-1);
    }

#ifdef COMPARE
    result_serial   = (int*) malloc(XPIX * YPIX * sizeof (*result_serial));
    for (row = 0; row < YPIX; row++) {
 im = IM_START + (row * (IM_STOP - IM_START)) / (YPIX - 1.0);
 for (col = 0; col < XPIX; col++) {
     re = RE_START + (col * (RE_STOP - RE_START)) / (XPIX - 1.0);
     iter = iterate(re, im, MAXITER);
     result_serial[row * XPIX + col] = iter;
 }
    }
#endif

    result_parallel = (int*) malloc(XPIX * YPIX * sizeof (*result_parallel));
#   pragma omp parallel for \
               private(row, col, re, im, iter) \
               shared(result_parallel) \
        schedule(dynamic, 1)
    for (row = 0; row < YPIX; row++) {
 im = IM_START + (row * (IM_STOP - IM_START)) / (YPIX - 1.0);
 for (col = 0; col < XPIX; col++) {
     re = RE_START + (col * (RE_STOP - RE_START)) / (XPIX - 1.0);
     iter = iterate(re, im, MAXITER);
     result_parallel[row * XPIX + col] = iter;
 }
    }
#ifdef COMPARE
    for (row = 0; row < YPIX; row++) {
```

```
 for (col = 0; col < XPIX; col++) {
     if (result_serial[row * XPIX + col]
  != result_parallel[row * XPIX + col] )
  return -1;
 }
    }
#endif

#ifdef USEVT
    VT_finalize();
#endif

    return 0;
}
```