High Performance Computing with Linux

Shimon Panfil, Ph. D.

Consultant on Physics and Mathematics

What is HPC

High Performance Computing (HPC) is the name given to massive numerical computations, originated from scientific and technical problems. Really this was the origin of all computers, but due to fast growth of business applications in 70's and home applications in 80's HPC became relatively small field. However, it is still alive.

HPC is always on the edge of computing technology, every new breakthrough expands the domain, transforms previously unsolvable problems to difficult ones and so on. Up to the middle of 90's, HPC was practically isolated from other parts of programming world. Most of (good) programming books and courses do not even mention HPC and most of (good) programmers know nothing about mathematics, numerical analysis etc. HPC was done by academic and industry research people, mainly in FORTRAN and on supercomputers. The vendors of these highly expensive machines also produced their own, highly optimized, compilers and libraries. All technicalities like pipelines, vectorization, multilevel memory organization, ...were hidden behind language constructions. I bet, most of HPC people do not know that, e.g. Cray is shared-memory vector-parallel processor system.

Due to well known political and economic processes in 90's, "supercomputing" lost most of its governmental funding. Therefore supercomputer vendors as independent entities become practically extint, e.g. Cray was purchased by SGI and HP swallowed Convex in 1996. So HPC now is forced to cope with COTS (Commodity Off The Shelf) hardware and software, fitting it for its purposes.

On the other hand 3D graphics and encrypted communications opened the door for HPC into the market which, of course, knew what was High Performance but not in this sense.

So the world of computing is united again.

Why Linux

There are 117 reasons for using Linux. Specifically for HPC the top 3 are:

1. Linux is made with all cutting-edge technologies. The most important motivation for people engaged in Linux is to do something the best in the world.

2. Its source codes are available, so you can understand what is going on and tune if necessary.

3. All the good stuff from UNIX world is available (gcc in particular).

HPC Cookbook

How to write HPC software in 7 simple steps:

1. Understand the problem at hand, formulate it in mathematical language.
2. Choose the most adequate algorithm.
3. Write software.
4. Test it.
5. Fix bugs.
6. Repeat two previous steps until the software is ready.
7. Give it to user.

Understanding of problem

When you are going to write, say, text editor, you probably know what it is supposed to do and why it will be better than existing ones. The situation may be quite different with HPC, where the problem may not be well defined. What your potential user tells you, may be not what he wants, which in turn may be not what he needs. Though this step has no direct relation to programming it is the most important one for HPC software.

Algorithms

Start from the simplest and most robust algorithm. If it is not efficient enough, search for a better one, (*better for your problem, not in general!*). Fortunately working on HPC you can safely assume that user really understand, what he is doing. Feel free to consult him at any stage.

Uniprocessor optimizations

Generally, a good compiler (like gcc) does good job for you, especially with your collaboration. Excess code, to much modularization and some "optimizations" can clutter your code and confuse the compiler. (Clutter is everything that contributes to runtime but not to answer.) It comes in to forms: overhead and compiler's flexibility restriction. Type conversions, e.g. contribute to overhead, and ambiguous pointers restrict flexibility by putting *fences* — places in program, where instructions appearing before and after can not be safely intermixed. Some constructions (subroutine calls, indirect memory references and tests within loops) contribute to both clatter forms.

It is a good idea also to use in-lining, unroll short loops within long ones, . . . , but **a better algorithm boosts performance much more than all those tricks.**

SMP vs PP

Symmetric Multi-Processing (SMP) refers to a large number of small independent tasks. This kind of load is typical for servers. High Performance SMP servers have 4–64 similar processors working with shared memory and performance depends practically linearly on the number of processors. Similar architecture with 1–4 processors is used in high end workstations.

HPC deals with another challenge: to make a task which takes, say, 16 hours to complete on 1 processor, to finish after running 1 hour on 16 processors. Sometimes it is simply impossible. Suppose that only half of the problem can be parallelized, so even with infinite number of processor only double speed is possible. This trivial consideration is known as Amdahl's law.

Misunderstanding of the principle difference between SMP and PP, makes a lot of troubles.

A Taxonomy of Parallel Architectures

The simplest taxonomy suggested by M.Flynn is based on single or multiple instruction and data stream:

SISD Single Instruction Single Data.

- single CPU WS

SIMD Single Instruction Multiple Data.

- array processor

MISD Multiple Instruction Single Data

- SMP (not exactly, but the closest)

MIMD Multiple Instruction Multiple Data

- General Multiprocessor

If we add memory type classification:

UMA Uniform Memory Access.

NUMA Non-Uniform Memory Access

DM Distributed Memory

and interconnection topology:

- 2D mesh
- toroid
- hypercube
- ...

We are ready to go ahead.

Not all combinations make sense and even less are used in practice, these are:

MISD UMA — SMP machines

MIMD NUMA — supercomputers

SIMD DM — supercomputers

MIMD DM —NOW (Network Of Workstations) or Clusters

Cluster is simply a dedicated network of workstations. Though it sounds good to harness the power of your colleague's WS or even spread your problem over the Internet all throughout the world, the problems of effectiveness, workload, bandwidth and security make cluster the only practical cost-effective solution for HPC.

PP programming

- Shared memory:
  Conceptually shared memory parallel (or concurrent) programming is almost trivial for unixoids. You write "usual" multiprocess or multi-thread program. Linux support SMP, it will affiliate process and threads with processors.
- Supercomputers:
  If you will meet one you will probably be taught how to use it.
- Clusters:
  Message passing programming, interesting, challenging, promising.

Message-Passing Environments

A message-passing interface is a set of functions and subroutine calls that allow to split your C or FORTRAN application for parallel execution. Data is divided and passed out to other processes as messages. In a sense this is 'assembler' for PP. You can get ultimate performance, but if not it might be your fault. Compiler and OS are completely unaware of parallelism. The two most popular environments are: Parallel Virtual Machine (PVM) from Oak Ridge National Lab., and Message Passing Interface (MPI) standard, which has a number of implementations, in particular Local Area Machine (LAM) from Ohio Supercomputer Center. Both PVM and LAM work on networks of computers with variety of architectures, including supercomputers and produce unified message-passing environment. PVM is reported to work on networks including NT and W95 machines.

PVM or LAM

PVM advantages:

1. flexibility;
2. inhomogenious network;
3. dynamical group (of processes) generation;
4. fault tolerance;
5. interlanguage communications.

MPI(LAM) advantages:

1. simplicity;
2. performance.

6 basic functions of MPI

It is possible to write non-trivial MPI program with only 6 functions:

1. MPI_Init(&argc, &argv);
2. MPI_Comm_rank(MPI_COMM_WORLD, &rank);
3. MPI_Comm_size(MPI_COMM_WORLD, &size);
4. MPI_Recv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &status);
5. MPI_Send(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD);
6. MPI_Finalize();

```
/*Laboratory for Scientific Computing
 * University of Notre Dame
 */
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
MPI_Status status;
int num, rank, size, tag, next, from;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
tag = 201;next=(rank+1)%size;from=(rank+size-1)%size;
if (rank == 0) {
```

```
    printf("Enter the number of times around the ring: ");
    scanf("%d", &num);
    printf("Process %d sending %d to %d\n",rank,num,next);
    MPI_Send(&num,1,MPI_INT,next,tag,MPI_COMM_WORLD);
  }
  while (1) {
    MPI_Recv(&num,1,MPI_INT,from,tag,MPI_COMM_WORLD,&status);
    printf("Process %d received %d\n", rank, num);
    if (rank == 0) {
        num--;
        printf("Process 0 decremented num\n");
    }
    printf("Process %d sending %d to %d\n",rank,num,next);
    MPI_Send(&num,1,MPI_INT,next,tag,MPI_COMM_WORLD);
    if (num == 0) {
```

```
        printf("Process %d exiting\n", rank);
        break;
    }
  }
  if (rank == 0)
    MPI_Recv(&num,1,MPI_INT,from,tag,MPI_COMM_WORLD,&status);
    MPI_Finalize();
    return 0;
}
```

References

[1] Kevin Dowd & Charles Severance,'High Performance Computing', 2 ed, O'Reilly, 1998

[2] Barry Wilkinson, Michael Allen,'Parallel Programming', Prentice Hall, 1999

[3] Bo Kågström, Jack Dongara, Erik Elmorth, Jerzy Wasniewski (Eds), 'Applied Parallel Computing' (Lecture Notes in Computer Science 1541) Large Scale Scientific and Industrial Problems, 4th International Workshop PARA'98, Umeå Sweden June 1998 (Proceedings)