Parallelisation of engineering codes - the pedestrian approach

Dr. Peter Junglas
TU Hamburg-Harburg
Computing Centre

December 17, 1996

1 Introduction

Much effort has been spent in parallelising large engineering applications like finite element or fluid dynamics codes. This is fully justified by their large distribution and wide area of applicability. Special emphasis has been laid on scalability, since these codes are often used for very large problems running on massively parallel computers.

The situation for many research codes in engineering sciences is very different: These programs are often used in only a few projects, and they are mainly run on workstations and the local compute server. Therefore parallelising is only considered if it's not too expensive, and scalability up to large processor counts is seldom an issue.

In the following we want to show that the more decent goal of achieving a reasonable speedup on a rather small number of processors is feasible with a reasonable effort, at least on a virtual shared memory system like the Exemplar. For this purpose we will present some codes, which were used or developed in scientific projects at the TU Hamburg-Harburg, and show, how they could be parallelised with little effort on up to one hypernode, using the standard SPP-UX programming tools.

Though it may be argued that this is not the 'high art' of parallelisation, it still gave the users a push in turnaround times, and even made some projects possible, which seemed unfeasible before. Moreover it shows that there is another, more evolutionary road to parallelism than spending years in redesigning, though leading to not quite as impressive results - the pedestrian approach.

2 Example Codes at the TU Hamburg-Harburg

2.1 CONCEPT

CONCEPT is a program for the simulation of electromagnetic scattering and radiation. It is mainly used for electromagnetic compatibility calculations, f.i. the effects of mobile telephones on the electric systems of a car. The first version was written in 1980 [1]. It has been used in many applications and is further developed at the Dep. of Theoretical Electrical Engineering, TUHH [5]

CONCEPT solves Maxwell's equations with a method of moments [3] leading to full matrices, which are not hermitian and usually ill conditioned. It's about 3 MB of FORTRAN77 source code plus several MBs for postprocessing tools. Typical runs need several hours and about 600 MB memory. Often there are lots of runs needed, with manual intervention in between.

The program consists of three steps:

```
preprocessing of input data
setup of system matrix
solving the system
```

The system matrix is N x N double complex ($N \approx 6000$, planned $N \approx 10000$), which is solved by an LU decomposition. Since this step dominates the cpu time for large N, the first step of the parallelisation was simple: Just replace the LU decomposition by the appropriate Veclib calls. This was done in two days, where the most time was needed for the (scalar) port to the SPP. The timing results for N = 6051 are:

	1 cpu [min]	6 cpus [min]	speedup
preprocessing and matrix setup	35.0	35.0	1.0
solver	67.5	15.0	4.5
total	102.5	50.0	2.1

For N=10000 the speedup will be 2.5 on 6 cpus. Not bad for so little an effort, but not quite satisfying. At the moment work on the parallelisation of the matrix setup is in progress, with promising preliminary results.

2.2 SELF

The program SELF computes tensions and shears in polycristalline materials. These parameters are used f.i. in the simulation of rolling of metal sheets for car production. The program was written in 1993 by Ricardo Lebensohn from the University of Rosario (Argentina) [4] and modified by Christian Hartig, Dep. of Physics and Technology of Materials, TUHH.

SELF uses a selfconsistency approach for the tensions of one grain in the field of all others. It consists of 100kB FORTRAN77. An example with 800 grains of very simple crystal structure, running 20 iterations, needed 32 h cputime and 70 MB on one cpu of the SPP1000, but realistic problems require much more grains and iterations.

Profiling with the CXpa showed that almost all cpu time is used in a loop over the grains. Since the iterations of this loop are independent, the parallelisation was straightforward: After streamlining the code by separating the critical loop in a routine and localising lots of global variables, it was enough to insert simple directives (LOOP_PARALLEL, LOOP_PRIVATE). The results of two days' work were convincing:

no. grains/iterations	1 cpu [min]	6 cpus [min]	speedup
24 /2	8.0	1.8	4.4
100 /2	28.7	4.8	6.0
800 /20	1920.0	320.0	6.0

For interesting problems the scalar part is completely negligible and the scaling perfect.

2.3 FDTD3D

FDTD3D is a program for the analysis of passive microwave circuits. It was written in 1995 by Rodrigo Tupynambá, Dep. of Microwave Engineering, TUHH [6, 7].

It uses finite difference methods with special boundary handling to solve Maxwell's equation. The code is 160 kB FORTRAN77. A typical run needs 22 h cpu time and 170 MB memory, but for the usual optimization problems lots of runs are necessary.

The program first partitions the circuit in cells and blocks, then computes the input gate parameters,

and finally runs the 3d finite difference solver. The cpu time is dominated by the last part, which has the following structure:

```
do t = 1, maxn
   call stepb
   call boundb
   call stepe
   call bounde
enddo
```

All four subroutines loop over all blocks:

```
subroutine stepb
integer q, qmax

do q = 1, qmax
  lots of work with not so large arrays
enddo
```

Since there are no dependencies inside the q loops, the first approach was simply to parallelize these loops. This didn't work at all (speedup < 1), since the overhead for thread creation and loop partitioning was too large for the very short loops (milliseconds).

The next attempt was to create the threads only once - before the outer (t) loop - using the cps library, and to distribute the q loops to the threads by hand. Since there are dependencies between the calls to stepe, bounde etc. several synchronisation calls had to be inserted.

Now the speedup was better, but still too low (about 2 on 6 cpus). This is due to bad load balancing caused by very different execution times for the iterations over the inner (q) loops. So we changed the distribution of the iterations to the threads from a block to a round-robin scheme, arriving at the following (pseudo-) code:

```
call spawn_threads(n_cpu)
call compute_distribution(tqmin, tqmax, tdq)

do n = 1, maxn
   call wait_barrier
   call stepb(tqmin, tqmax, tdq)
   call wait_barrier
   call boundb(tqmin, tqmax, tdq)
   call wait_barrier
   call stepe(tqmin, tqmax, tdq)
   call wait_barrier
   call bounde(tqmin, tqmax, tdq)
enddo

call join_threads(n_cpu)
```

where the routines were modified accordingly:

```
subroutine stepb(tqmin, tqmax, tdq)
integer q, tqmin, tqmax, tdq

do q = tqmin, tqmax, tdq
   lots of work with not so large arrays
enddo
```

After five days we finally got satisfying speedups:

	1 cpu [min]	6 cpus [min]	speedup
scalar parts	80	80	1.0
parallel loops	1240	220	5.6
total	1320	300	4.4

Larger problems containing more blocks lead to a reduction of the scalar part and to even better load balancing. Parallelisation of the scalar rest should be relatively easy, since it's structure is similar to the parallel part, but it will not be done! The user was satisfied with the results, did lots of runs - and the project is finished.

2.4 PROGNOSIS

The last example is not a stand-alone program but a MATLAB [1] application. It's purpose is the prediction of electric power requirements on board of ships, with a prediction time of 5 to 60 minutes. It uses MATLAB and a neural network toolbox [2] to train different types of networks. The program was written in 1996 by Stephan Busch, Dep. of Marine Engineering and Energy Systems, TUHH [2].

For the training of one network one needs several days of cputime and 700 MB to 1.1 GB of memory. Furthermore many different parameters and kinds of neural networks have to be compared. Therefore the need for a faster (parallel) solution was clear. Unfortunately MathWorks states explicitly [3] that they will not port MATLAB to parallel machines.

Analysis of the cputimes showed that almost all time is spent in the MATLAB matrix multiplication and LU decomposition routines. Since MATLAB is able to dynamically call user routines - so called MEX files - like ordinary MATLAB commands, it should be possible to replace the critical calls by own ones, which in turn use the parallel veclib routines.

One problem remained: MATLAB is an HP-UX binary (SOM format) and cannot dynamically bind parallel SPP binaries (ESOM objects). We solved this by using two different programs: a master in SOM format and a slave in ESOM format. The master is called by MATLAB and communicates with the slave process via System V shared memory. The slave does the parallel veclib computations. Alltogether we get three parts:

```
Matlab:
   ...
   matlab commands
   ...
   result = mymat(input);
   ...

Master (mymat.c):

  get input from matlab
  get shared memory segment
  copy data into shared memory
  call slave
  copy result from shared memory into MATLAB area
  clean up
```

[1] MATLAB is a product of The MathWorks Inc.

[2] The Neural Network Based System Identification Toolbox is written by Magnus Norgaard, Technical University of Denmark

[3] cf. http://www.mathworks.com/solutions/303.html

```
Slave (slave.c):

  attach shared memory segment
  do parallel computation
      (all data in shared memory)
  clean up
```

The following table shows the timing results of the matlab-master-slave program as speedups compared to the original MATLAB code:

	1 cpu	6 cpus	parallel speedup
matrix multiplication (n=2000)	22	98	4.5
lu decomposition (n=2000)	5	14	3.0
complete program	15	60	4.0

The parallel speedup is nice, but even more impressive is the speedup using veclib on one cpu. This shows that the MATLAB matrix operations on HP-UX/SPP-UX are extremely slow.

3 Conclusions

Looking back at these - and some other - parallelisation projects, there are some simple lessons, we have learned:

- To find the hot spots it is important to analyse large program runs. Small test examples are often misleading.
- Streamlining of the critical code segment is an important step before one can decide which parallelisation strategy to use.
- There is a large range of parallelisation methods to choose under SPP-UX: From message-passing over System V communication to shared memory directives and explicit thread programming.
- Most users are interested in only one thing: Getting their results faster than before – including the time to parallelize the program. Except for very often reused programs this means: If it cannot be done quickly, it will not be done!

References

[1] H.-D. Brüns. *Pulserregte elektromagnetische Vorgänge in dreidimensionalen Stabstrukturen.* PhD thesis, Universität der Bundeswehr Hamburg, 1985.

[2] Stephan Busch. Work in progress.

[3] R.F. Harrington. *Field Computation by Moment Methods.* The Macmillan Company, New York, 1968.

[4] R. A. Lebensohn and C. N. Tomé. A self-consistent anisotropic approach for the simulation of plastic deformation and texture development of polycrystals: Application to zirconium alloys. *Acta Metall. Mater.*, 41:2611, 1993.

[5] Th. Mader. *Berechnung elektromagetischer Felderscheinungen in abschnittsweise homogenen Medien mit Oberflächenstromsimulation.* PhD thesis, Technische Universität Hamburg-Harburg, 1992.

[6] Rodrigo C. Tupynambá and Abbas S. Omar. Some improvements to the fdtd algorithm for the analysis of passive circuits. In *1995 IEEE MTT-Symposium Digest*, pages 1657 – 1660, 1995.

[7] Rodrigo C. Tupynambá and Abbas S. Omar. The synthesis of passive circuits using the fdtd. In *1996 IEEE MTT-Symposium Digest*, pages 749 – 752, 1996.