Comparison of Parallel Simulation Techniques
ACSL
Shared Memory Multiprocessors/UNIX

Shared memory multiprocessors hold great potential for improvement of simulation execution times, owing to zero-latency communication and the absence of a need to encapsulate or otherwise prepare data for transmission. Owing to its FORTRAN heritage, ACSL relies heavily on static data, i.e. data not kept on a stack. It is not possible to create *instances* of ACSL models dynamically at run-time. Thus, a thread-based implementation (where all processes share one address space) is less appropriate than a true fork process based implementation with an explicitly allocated shared memory area.

We distribute the work among the processors (using **fork**) by partitioning the equation set among the processors. Since the equations are not independent, the values of the state variables must be exchanged from time to time. Ideally, state variables would reside in the shared memory segment and thus would be visible to all processors immediately. ACSL does not allow state variables to be placed at arbitrary memory locations; thus, the programs must copy the state variables that are to be shared into the shared memory segment. A further difficulty arises because the processors of multiprocessor machines are not guaranteed to remain synchronised. Consequently a synchronisation phase must be executed during every update.

Two different types of synchronisation were tested: one using operating system semaphores and one that works exclusively in user mode. For this strategy to work, all processes must be integrating at the same speed - which is guaranteed by using fixed step algorithms.

A library (with two versions) was developed for forking the processes and exchange data. The first version of the library, the one using operating system semaphores (S_LIB), consists of three user-visible procedures:

shminitialize takes two parameters: the number of processes to create - typically the number of processors in the machine - and the number of bytes a process needs in shared memory. It returns a small integer indicating the processor number this process is running on; user code must later use this number to select the appropriate right-hand sides of the differential equations. **shmwriteread** takes a vector of values to be exchanged with the other processors and a vector where the values calculated by other processors are returned. **shmwriteread** is typically called in the DYNAMIC or DISCRETE SECTION, i.e. once per communication interval or for each evaluation of the right-hand side of the differential equations. **shmterminate** releases the resources allocated by **shminitialize**. After **shmterminate**, only the original process remains.

Besides needing the operating system to synchronise, the above version suffers because it copies data unnecessarily. The second version of the library (F_LIB) avoids this copying completely by making the shared memory area visible to the user's code. In case of the HP Precision Architecture RISC machine (used for these investigations, with four processors) the only way to do this in FORTRAN is by passing the address of the shared memory area as an actual parameter to user-supplied subroutines.

The interface changes as follows: **shminitialize** additionally takes two subroutine arguments. The first is called when data is to be copied into the shared memory segment, the second is called to read from the shared memory segment. Instead of **shmwriteread**, this second version of the library calls **shmbarrier** (using subroutines made available in the **shminitialize** function).

Monte Carlo-Study

The first version of the library was used to solve this "Monte- Carlo Study" task. In the INITIAL SECTION M parallel processes (**procs**) are forked with **shminitialize** performing 1000 / M simulation runs. No data transfer is necessary during the simulation. In the TERMINAL SECTION the results are read out by **shmwriteread**. The following table shows the efficieny factor f of the parallelisation.

```
INITIAL; .......
proc= shminitialize(procs, resultnum*4); ...
LOOP.. i = i + 1; ..........
END
DYNAMIC
   DERIVATIVE; ............
TERMINAL; ...........
call shmwriteread (procresults = results)
if .... goto LOOP
call shmterminate; .........
END
```

	M = 1	M = 2	M = 3	M = 4
S_LIB	1	0.535	0.345	0.257

Partial Differential Equation

Discretising the PDE with the method of lines (with N lines) results in a system of weakly coupled ordinary differential equations (2N equations). Each process has to integrate 2N / M equations, and data transfer is only necessary between the boundary lines. The parallel processes are forked in the INITIAL SECTION, the data transfer takes place in the DYNAMIC section each communication interval ($c_{int} = h$, with S_LIB, F_LIB in brackets):

```
INITIAL; ...........
pr = shminitialize (procs, boundaries * 4)
 [ pr = shminitialize (procs, boundaries,
                       srupdate, sr read) ]
END
DYNAMIC
DERIVATIVE; ..............; END
call shmwriteread (boundary  =  bound)
 [ call shmbarrier ]
END
TERMINAL; call shmterminate: END
```

Fourth order Runge-Kutta was used to integrate the equations, which evaluates each equation four times per integration step - whereas data is exchanged only once. To eliminate the resulting discretisation error, **shmbarrier** was called in the DERIVATIVE SECTION; surprisingly, execution speed increased in some cases.

Results in term of the efficiency factor *f* for different numbers of processors M and different numbers of lines N are summarised in the following tables:

N=600	M = 1	M = 2	M = 3	M = 4
S_LIB, DYN.S.	1	0.849	0.472	0.400
F_LIB, DYN.S.	1	0.578	0.450	0.342
F_LIB, DER.S.	1	0.729	0.458	0.345

M = 4	N = 600	N = 800	N = 1000
S_LIB, DYN.S.	0.439	0.400	0.375
F_LIB, DYN.S.	0.361	0.342	0.322
F_LIB, DYN.S.	0.357	0.342	0.319

Coupled Predator-Prey System

This task consists of only five pairs of equations that are strongly coupled. As only four processors were available, and the forked processes have to be of the same structure, only three processors were used; each processor calculating two pairs. Forking and data transfer was done as in the PDE task.

Parallelisation is not successful. It turned out, that the communication overhead is as big as the benefit of parallelisation. If a communication interval c_{int} bigger than the stepsize h is chosen ($c_{int} = \alpha h$, $\alpha > 1$), a speedup can be achieved. The systems remains stable until $c_{int} = 20h$. The following table summarises the results for the efficiency factor *f*, where also a version with two processors (each three pairs of models) was tested:

M=3 F_LIB c_{int}=h	M=3 F_LIB c_{int}=2h	M=3 S_LIB c_{int}=20h	M=3 F_LIB c_{int}=20h	M=2 F_LIB c_{int}=5h
1	0.943	0.917	0.800	0.813

K. Schwarz, F. Breitenecker, ARGESIM, Dept. of Simulation Techniques, TU Vienna, Wiedner Hauptstr. 8-10, A-1040 Wien, Email: argesim@argesim.tuwien.ac.at