Parallel Processing Issues [2]

Parallel processing is a method of splitting a computational task into subtasks, and then simultaneously performing the subtasks. This section discusses the different ways parallel processing strategies used on UNICOS systems, and what effects those implementations have on the performance of your code.

2.1 Parallel Processing Overview

Parallel processing is performed at the hardware level, the operating system level, and the code level. This subsection briefly discusses these different types of parallel processing.

2.1.1 Parallel Processing: Hardware Level

At the hardware level, parallel processing is accomplished by using the following methods:

- *parallel instruction execution*, which is the execution of one instruction per clock period, even those instructions that take several clock periods to complete execution.
- *vectorization,* which is a form of parallel processing that uses instruction segmenting and vector registers.
- *I/O subsystems* or *foreground processors*, which is the execution of operations in parallel with processes running in the main processors that perform I/O.
- *time slicing*, in which the system works on several jobs or processes simultaneously.
- *pipelining*, which allows each step of an operation to pass its result to the next step after only one clock period (see Figure 1).
- *chaining*, which allows the movement of elements to continue from one vector operation to another, so that a process including more than one vector operation is executed as one long vectorized operation (see Figure 2).

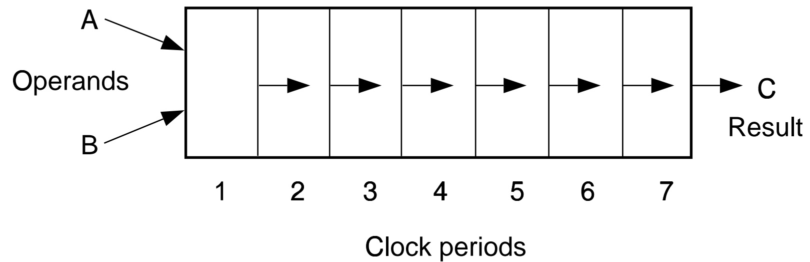

Figure 1. Pipelining in add operation

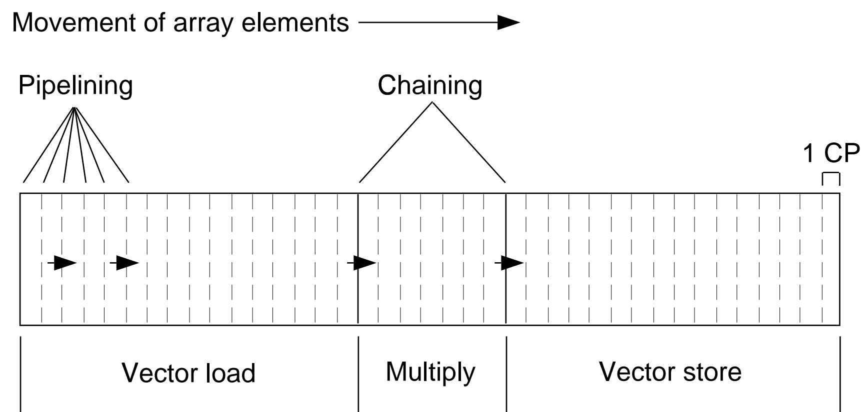

Figure 2. Pipelining and chaining

2.1.2 Parallel Processing: Operating System Level

At the operating system level, parallel processing is accomplished by using the following methods:

- *multiprogramming*, which occurs when a processor switches between the jobs or processes in the system.
- *multiprocessing*, which occurs when processors simultaneously work on as many programs as there are processors.

- *multitasking*, which is a general term describing more than one processor working to complete a single program. This term has become synonymous with parallel processing.

2.1.3 Parallel Processing: Code Level

There are several ways to alter user code to take advantage of parallel processing:

- *macrotasking* lets you take advantage of multitasking at the subroutine level by inserting library calls to express parallelism. This works well on programs that can be explicitly partitioned into tasks and can be simultaneously executed on multiple processors.
- *microtasking* allows you to insert directives at the loop or block level of the code to indicate sections that can be executed on multiple CPUs.

Macrotasking and microtasking are seldom used anymore. *Autotasking* is the preferred method to use for parallel processing at the code level. Like microtasking, Autotasking exploits parallelism at the loop or block level of code. It has lower overhead than microtasking, and it can be made fully automatic. It does not require any direct user intervention, but users can interact with the system via directives and switches.

The Scientific Library routines were designed to be fully optimized to take advantage of parallelism and to automatically use Autotasking during execution.

When using the CF90 compiler, you can use the `f90 -O3` command. This command invokes aggressive optimization and Autotasking. One of the optimization actions the compiler then performs is to substitute Scientific Library routines (where appropriate) in the code. The routines are called at an entry point where they will use less compiling time.

See the *CF90 Commands and Directives Reference Manual* or the `f90`(1) man page for more information about using the `f90` command.

2.2 Costs and Benefits of Parallel Processing

Parallel processing can eliminate idle CPU time because the workload is divided among all CPUs; therefore, the amount of work performed per unit time (the *throughput*) increases. However, parallel processing also introduces some overhead into program execution. In some cases, you may be able to

reduce wall-clock time, but at the cost of extra CPU time which increases because more machine resources are used.

This subsection discusses these benefits and some of the costs of using parallel processing.

2.2.1 Benefits of Parallel Processing

By using parallel processing, you can alleviate some of the following common problems:

- Maximum-memory jobs: if the memory is occupied by a few large-memory jobs, one or more of the CPUs might be idle even though there are other jobs to run.
- Dedicated machine: if the computer is running a single job, then all other CPUs are idle.
- Light workload: if the amount of jobs waiting for a CPU is less than the total number of CPUs, then one or more of the CPUs becomes idle.

With parallel processing, the additional CPUs reduce the wall-clock time instead of sitting idle. Even when very little idle time exists, using additional CPUs can still lead to benefits.

2.2.2 Parallel Processing Overhead

Parallel processing introduces some overhead into program execution. This subsection discusses some of the common types of overhead introduced by parallel processing:

- Multitasked programs require more memory than unitasked programs, and they can contain more code, more temporary variables, and can require additional stack space.
- Multitasked jobs can be swapped more often, and remain swapped longer, on a heavily loaded production system.
- Processors are forced to wait on semaphores during the process of synchronization.
- Overhead is incurred when slave processors are acquired (on entry to a parallel region) and at synchronization points within parallel regions. Tests show that the overhead of executing extra Autotasking code adds a nominal 0% to 5% to the overall execution time.

- If inner-loop Autotasking is used, vector performance can decrease because of shorter vector lengths and more vector loop startups.
- Processors are sometimes held for the next parallel region to improve efficiency. While holding a processor can save time, it also costs time to acquire and hold them.

Because overhead is associated with work distribution, jobs with large granularity have less partitioning than smaller jobs. Large jobs, however, may have problems with load balancing.

2.2.3 Parallelism and Vectorization

A multitasked program usually has the same amount of work for the CPUs to perform as does a similar unitasked program. However, when the work is spread across many processors, the wall-clock time required to complete the work is usually less.

Vectorization is a form of parallel processing in which array elements are processed by groups. Vectorization usually decreases both CPU time and wall-clock time; multitasking decreases only wall-clock time. Thus, vectorization can give you large gains in terms of saving time and have relatively low costs associated with it.

If you make direct calls to Scientific Library routines, your CPU time or wall-clock time could also be affected. Scientific Library routines take advantage of vectorization and multitasking; because of the costs associated with multitasking, you may see an increase in CPU time when using these routines. You can control this increase somewhat by using the `NCPUS` environment variable to control the maximum number of CPUs to be used on a job. See Section 2.4.1, page 12, for details about using environment variables.

When using multitasking, the timing results in a nondedicated environment are not always reproducible. Several factors can affect the timings obtained from multitasked routines, including some of the following:

- System load
- I/O performed by other jobs
- Memory contention

If accurate timing results are desired, it is best to run a job in a dedicated environment where it is the only job being run.

2.2.4 Parallelism and Load Balancing

The parallelism for any region is determined by the number of partitions, or *chunks*, of independent work the region contains. If an autotasked loop has N iterations, N is the extent of parallelism for that loop. This means that the loop can effectively be broken into N independent chunks of work. For autotasked inner loops, the extent of parallelism is the number of iterations divided by the vector length when both Autotasking and vectorization can be used.

Load balancing is the process of dividing work done by each available processor into approximately equal amounts. In a multiuser environment, the number of available processors is constantly changing. When Autotasking is used, it automatically tries to perform dynamic load balancing by creating small-granularity parallelism. For example, if a `DO` loop is autotasked, work is allocated by default to each task one iteration at a time.

A direct relationship exists between load balancing and the extent of parallelism:

- The higher the extent of parallelism, the easier it is to balance the workload evenly across the processors.
- Small-granularity parallelism is easier to balance across available processors than large granularity parallelism.
- Small-granularity parallelism generates more overhead than large granularity parallelism. Synchronization is required each time a chunk of work is allocated to a processor.

2.3 Performance Issues

This subsection discusses the following different aspects used to define the performance of parallel processing: speedup, efficiency, and percentage of parallelism in a program.

2.3.1 Calculating Speedups in Multitasked Programs

On a dedicated (nonproduction) system, the speedup ratio for a multitasked program can be calculated in the following manner, where T_1 is the wall-clock execution time on a single-processor and T_p is the wall-clock execution time on p processors:

$$\text{Speedup ratio} = \frac{T_1}{T_p}$$

(2.1)

With p CPUs, a speedup ratio as close as possible to p is desired. If the program were completely parallel and no overhead existed, the speedup would be equal to p (for details about the overhead associated with multitasking, see Section 2.2.2, page 6).

For example, suppose a job takes 8.7 seconds to run on a single processor. When the job is rerun using four processors, the execution time decreases to 2.5 seconds; the speedup is the following:

$$s = \frac{8.7}{2.5} = 3.48$$

(2.2)

The speedup ratio of multitasked code has two limitations: Amdahl's Law (representing the speedup related to the sequential portion of the code), and multitasking overhead (discussed in Section 2.2.2, page 6).

2.3.1.1 Amdahl's Law

Some sections of programs, known as *single-threaded code segments,* must use a single processor. Amdahl's Law for parallel processing illustrates the effect of single-threaded code segments on multitasking performance. Amdahl's Law is shown in the following equation:

$$s = \frac{1}{(1 - f) + \frac{f}{p}}$$

(2.3)

The following components appear in this equation:

- s: Maximum expected speedup from multitasking
- p: Number of processors available for parallel execution
- f: Fraction of a program that can execute in parallel (the *parallel fraction)*

For example, suppose that code is 98% parallel, implying that 2% of the code runs in serial mode. Suppose that 64 processors are available. According to Amdahl's Law, the maximum speedup that you can expect is:

$$s = \frac{1}{(1 - 0.98) + \frac{0.98}{64}} = 28.32$$

(2.4)

The speedup from multitasking, s, is in terms of wall-clock time, not CPU time. Speedups equal to the physical number of processors require that the executing program use all processors effectively 100% of the time with no overhead. Because this is not possible, performance is dominated by the fraction of the time spent executing serial code.

Maximum speedup depends on your serial code. If the number of processors were infinite, the parallel term would be 0, but the serial term would remain, giving a maximum possible speedup of the following:

$$s = \frac{1}{.02} = 50$$

(2.5)

This is sometimes interpreted to mean that only 50 processors can be used on a 98% parallel problem. You can use any number of processors, but because the maximum possible speedup is a constant, the efficiency decreases as the number of processors increases.

The `amlaw` command displays the maximum theoretical speedup when you provide the number of CPUs and the percent parallelism. See the `amlaw`(1) man page for information about using the command.

2.3.2 Determining Efficiency

The efficiency of multitasked code is a comparison of the measured speedup with the maximum possible speedup. The overall efficiency of some code may be low because some parts of the code are inefficient; this will affect the overall efficiency.

The efficiency is defined in the following formula, where p is the number of processors, s is the measured speedup, and e is the efficiency:

$$e = \frac{s}{p}$$

(2.6)

If the program were completely parallel, and no overhead existed, the efficiency would be 1.0 (100%). As an example, suppose that the measured speedup running on four processors, compared to one processor, is s = 3.48. The efficiency is then defined as follows:

$$e = \frac{3.48}{4} = 0.87 = 87\% \tag{2.7}$$

2.3.3 Estimating Percentage of Parallelism

Another useful way to measure the efficiency of code is to measure the percentage of parallelism (the *parallel fraction* in Amdahl's Law). To estimate this figure, measure the actual time required to run a program on a single processor; also measure the actual time required to run the program in parallel on p processors. The ratio between the two times is the speedup, s. After s and p are known, the equation for Amdahl's Law can be rewritten for f as follows:

$$f = \frac{1 - \frac{1}{s}}{1 - \frac{1}{p}} \tag{2.8}$$

For example, suppose the measured speedup running on four processors compared to one processor is s = 3.48. The equivalent parallel fraction is the following:

$$f = \frac{1 - \frac{1}{3.48}}{1 - \frac{1}{4}} = 0.95 \tag{2.9}$$

This program thus achieves 95% parallelism, assuming the idealized model of Amdahl's Law.

2.4 Parallel Processing Environments

UNICOS systems have two basic parallel processing environments: the dedicated environment and the multiuser environment. This subsection provides overview information about these environments, as well as

information about the different environment variables you can set which can help you tune your system's performance.

Note: The settings for the different environment variables can significantly affect results and timings of code. See your system administrator for information about your site's use of these environment variables.

2.4.1 Environment Variables

You can use environment variables to predefine some characteristics of your shell; these environment variables are taken from the execution environment, and the values defined for the variables can affect your parallel processing environment.

The following environment variables allow you to tune the system for parallel processing without rebuilding libraries or other system software:

- `NCPUS`: Specifies the maximum number of CPUs available to work on a program. The default is 4 for machines with more than 4 CPUs. For machines with 4 or fewer CPUs, the default is the number of physical CPUs. The default of 4 CPUs was chosen for multitasked programs because experience has shown that in a nondedicated environment, this number often gives the best performance for a large number of programs.

- `MP_DEDICATED`: Specifies the type of machine environment. If set to 1, it specifies that you are the only user on the system. This allows the multitasking library to schedule differently to take advantage of the dedicated environment. If `MP_DEDICATED` is set to 0 or is not set at all, slave processors return to the operating system after waiting in user space.

  If you set `MP_DEDICATED` to 1 in a nondedicated system throughput can be degraded.

- `MP_HOLDTIME`: Specifies the number of clock periods to hold a processor before giving up the CPU when no parallel work is available.

Both `NCPUS` and `MP_DEDICATED` are read by Scientific Library routines, which are greatly affected by the settings for these variables. These routines use different strategies depending on the settings. This can result in different performance numbers.

The following list contains suggested settings for these variables, depending on the system load in a non-dedicated environment:

- Heavily loaded: set `MP_DEDICATED` to 0 and `NCPUS` to 1.

- Normal load: set MP_DEDICATED to 0 and NCPUS to 0.25 of the maximum CPUs available.
- Lightly loaded: set NCPUS to the maximum number of CPUs available, essentially treating the machine as a dedicated machine.

2.4.2 The Dedicated Environment

A dedicated work environment is often used in situations in which wall-clock time is as important as CPU time/usage.

To select parallel processing in the dedicated environment, set MP_DEDICATED to 1, and change the NCPUS environment variable to equal the total number of available processors. This ensures access to a number of processors equal to the value in NCPUS for the entire time your code is executing.

A dedicated environment is often used in the following ways:

- When a job is the only one running on the system. In this case, set NCPUS equal to the total number of available physical processors.
- When a job requires a large amount of memory; therefore, few, if any, jobs can be active at the same time. In this case, set NCPUS equal to the total number of physical processors available if the job requires all of central memory, or to a smaller number if less memory is required.
- When priority schemes are used to favor specific jobs.

The following assumptions are made about dedicated environments:

- The number of processors specified in NCPUS are available to users at all times.
- Users want to minimize wall-clock time even if it increases cumulative CPU time.

2.4.3 The Multiuser Environment

In a multiuser work environment, CPU time is often of more importance than wall-clock time. To select parallel processing in a multiuser or production environment, ensure that the MP_DEDICATED environment variable is set to the default (0); if MP_DEDICATED is not set, a multiuser environment is assumed. MP_DEDICATED is actually a tuning parameter that the multiprocessing library uses. Its value determines the strategy that the library uses for attaching and detaching processors to a job.

In the multiuser environment, you cannot control the number of processors that are attached to a job except to specify a maximum number by using the NCPUS environment variable. The actual number of processors that will be used cannot be known in advance and may change as the job runs. When working in a lightly loaded, multiuser environment, set NCPUS equal to a small number of the available physical processors.

The following assumptions are made about the multiuser environment:

- Users do not know how many processors will be available to a job during run time, except that the number will be less than or equal to NCPUS.
- It is probable that fewer processors than the number specified in NCPUS will be attached to a job; therefore, you should use a partitioning strategy that gives the best **average** performance.

2.5 Measuring Scientific Library Performance

This subsection contains information that will help you measure the performance of Scientific Library routines in your code. See *Optimizing Application Code on UNICOS Systems*, for additional information about measuring code performance.

2.5.1 Simple Measurements

You can use several simple measurements to check the performance of your code.

A simple way to measure performance is to use dedicated resources to perform your single CPU measurements. This ensures that all of the CPU time accumulated to your job is directly related to the work being done by your code (rather than CPU time that your job accumulates as a result of any swapping that occurs).

However, you can still obtain good measures of performance on a nondedicated machine (especially one that is only slightly busy) by using the SECOND(3F) library routine. This routine obtains the elapsed user CPU time since the program started. The CPU time returned by SECOND does not include system CPU time.

You must use dedicated resources to obtain accurate wall-clock time measurements when using parallel processing. Running tests on even a minimally loaded machine will show variations in parallel processing

performance because of the unpredictability of obtaining CPU time from the operating system.

You can also use the RTC(3F) library routine to measure the performance of routines. RTC returns the value of the real-time clock register. This register contains the number of clock ticks. To convert this value to seconds multiply the RTC or IRTC value by the number of cycles per second. To obtain the cycle time of UNICOS machines, use the target(1) command or the target system call using the following C function:

```
#include <sys/target.h>
float GET_CTICKS()
/* Return the clock period in picoseconds */
{
 struct target data;
        target(MC_GET_TARGET, &data);
        return data.mc_clk;
}
```

Then, to convert RTC to seconds, use the following command:

```
walltime = RTC() * GET_CTICKS() * 1.0e-12
```

2.5.2 Determining Multitasking of Routines

If you are trying to measure performance as a problem size, you should realize that not all Scientific Library routines use parallel processing. For small problem sizes, a routine might not use multiprocessing at all, but it will make use of multiple CPUs for larger problem sizes.

In the following example, the SGEMV(3S) routine is used to compare single CPU performance to multitasked performance. The performance measurement is done in loop 1000. The measurement technique used results in inaccurate measurements for small problem sizes because SGEMV does not multitask for problem sizes below a certain threshold.

Prior to the measurement loop, the data is initialized in a loop that is parallelized by the call to the CINIT subroutine. Because of the cost of acquiring multiple CPUs to apply to a single process, the library routines check the MP_HOLDTIME environment variable (this specifies the time that a process should hold on to acquired CPUs after the parallel region that requested those CPUs has completed). Those acquired CPUs may still be attached when measuring the SGEMV routine; if the problem size is below the SGEMV

multitasking threshold, the CPU time accumulated by these CPUs is included in the timing of the work done by SGEMV.

This same false CPU usage can occur if the problem size in the loop is decremented from large problems to small problems. In this case, the test that measures performance in the problem area around the SGEMV multitasking threshold is affected by the CPUs that are not being applied to the problem but are just waiting for their MP_HOLDTIME to expire. The holding of CPUs between parallel regions benefits those larger problem size test cases because the multitasking library does not need to request additional CPUs for the parallel region if they are already attached and waiting. Setting MP_HOLDTIME to 0 causes idle processes to be immediately released and results in more accurate measurements of the work being done.

TESTCASE

```
#!/bin/sh
maxcpus='4 8'                # max NCPUS
xgf1=sgemv_1.xgf             # results: xgraph input file
xgf2=sgemv_2.xgf             # results: xgraph input file
#
cat > mmstr.f << EOF
      PROGRAM MMSTR
      PARAMETER  (N =1024)
      REAL*8  A(N,N),B(N,N),C(N,N)
      REAL OPS, MFLOPS
      REAL CYCLET, GET_CTICKS
      INTEGER LOOPCNT
      INTEGER T1, T2, IRTC
      EXTERNAL SECOND, IRTC, GET_CTICKS

      CYCLET = GET_CTICKS() * 1.0e-12

         NSTART = N
         NEND = 512
         NINK = -128
 999     CONTINUE
C
      DO 1000 NIND=NSTART,NEND,NINK
        IF (NINK.LT.-8) THEN
          NST1 = MIN(NIND+8,N)
          NEN1 = MAX(NIND-8,8)
        ELSE
          NST1 = NIND
```

```
            NEN1 = NIND
          ENDIF
          DO 1000 NPART=NST1,NEN1,-8
             LDA = NPART
             LDB = NPART
             LDC = NPART
             CALL EX312(A,LDA,NPART)
             CALL EX312(B,LDB,NPART)
C
             CALL CINIT(C,LDC,NPART)
             LOOPCNT = NPART

C
              T1 = IRTC()
              S1 = SECOND ()
              DO I=1,LOOPCNT
                CALL SGEMV
('N',NPART,NPART,1.,A,LDA,B(1,I),1,0.,C(1,I),1)
              END DO
              S2 = SECOND ()
              T2 = (IRTC() - T1)/NPART
C
              OPS = 2*(NPART * NPART)
              WRITE(66,'(I5,E12.4)') NPART,S2-S1
              WRITE(77,'(I5,E12.4)') NPART, OPS/(T2 *CYCLET)
C
1000     CONTINUE
C
           IF (NINK.EQ.-128) THEN
              NINK = -64
              NSTART = NEND + NINK
              NEND   = 256
              GOTO  999
           ELSEIF (NINK.EQ.-64) THEN
              NINK = -32
              NSTART = NEND + NINK
              NEND   = 128
              GOTO  999
           ELSEIF (NINK.EQ.-32) THEN
              NINK = -8
              NSTART = NEND + 2 * NINK
              NEND   = 8
              GOTO  999
```

```
         ENDIF
C
 64      FORMAT(3X,I4,2X,'I',5(1X,E12.4,3X,'I'))
2000  CONTINUE
      END
C
      SUBROUTINE EX312(A,LDA,NPART)
      INTEGER NPART,I,J,NP1I,LDA
      REAL*8  A(LDA,NPART)
C
      DO 10 J=1,NPART
         DO 10 I=J,NPART
           NP1I=NPART+1-I
           A(I,J) = DBLE(NP1I)
           A(J,I) = DBLE(NP1I)
 10   CONTINUE
      RETURN
      END
C
      SUBROUTINE CINIT(C,LDC,NPART)
      INTEGER NPART,I,J,LDC
      REAL*8  C(LDC,NPART)
C
      DO 10 I=1,NPART
         DO 10 J=1,NPART
           C(I,J) = 0.0
 10   CONTINUE
      RETURN
C
      END
EOF
#
#  Get clock ticks to compute wall clock seconds
#
cat >get_cticks.c << EOF
#include <sys/target.h>
float GET_CTICKS()
/* Return the clock period in picoseconds */
{
 struct target data;
        target(MC_GET_TARGET, &data);
        return data.mc_clk;
}
```

```
EOF
#
cc -c -g get_cticks.c

cf77 -Zp get_cticks.o mmstr.f

#    get sgemv version

# initialize xgraph file
echo 'Device: Postscript' > $xgf1
echo "TitleText: `date '+%d-%h-%y %H:%M'` `uname -snmr`SGEMV" >> $xgf1
echo 'XUnitText: matrix dim N' >> $xgf1
echo 'YUnitText: Overhead [%]' >> $xgf1
echo 'YLowLimit: 0' >> $xgf1

# initialize xgraph file
echo 'Device: Postscript' > $xgf2echo "TitleText: `date'+%d-%h-%y
%H:%M'` `uname -snmr` SGEMV" >> $xgf2
echo 'XUnitText: matrix dim N' >> $xgf2
echo 'YUnitText: MFLOPS' >> $xgf2

NCPUS=1 ./a.out
mv fort.66 x.1
#
#  Elinimate Residual CPU usage by setting
#  HOLDTIME to 0
#
MP_HOLDTIME=0
export MP_HOLDTIME
for n in $maxcpus
do
   ja
   NCPUS=$n ./a.out
   ja -ct
   echo '\n" '$n CPUs'"' >> $xgf1
   echo '\n" '$n CPUs'"' >> $xgf2
   paste x.1 fort.66 |
   awk '{ ov = ($4 - $2)*100/$2; print $1, ov }' >> $xgf1
   cat fort.77 >> $xgf2
done
#
rm -f x.1 fort.66
```

```
test $DISPLAY && xgraph $xgf1 &
test $DISPLAY && xgraph $xgf2 &
```

2.6 Parallel Processing Strategies

This subsection describes the parallel processing strategies that the Scientific Library routines use in dedicated and multiuser environments. For small or very large problems, the strategies in both environments are the same. For medium-sized problems, however, the strategies differ greatly. The Scientific Library automatically implements these strategies without the need for user intervention.

2.6.1 Problem Sizes

The strategies presented in the following subsections apply to fine-grain data parallel problems. On UNICOS systems, this type of parallelism is effectively used by vectorization on one vector processor and parallelization across several vector processors.

The following is a classification of problem sizes:

- *vector problem* (VP) – Problems large enough for vector processing, but too small for parallel processing.
- *small parallel/vector problem* (SPVP) – Problems large enough for vector and parallel processing, but for which parallel processing degrades vector performance.
- Medium parallel/vector problem (MPVP) – Problems large enough for optimal vector and parallel processing, but for which load balancing can be a significant problem if a processor is lost.
- *large parallel/vector problem* (LPVP) – Problems large enough for optimal vector and parallel processing for which load balancing is not a significant problem. In this case, enough work exists to partition the problem into many subproblems so that the effect of losing a processor is minimized.

The boundary between the VP and SPVP classes is independent of the number of processors. However, the other boundaries between classes depend on the number of processors you try to use. A problem size of class LPVP on 2 processors could be of class MPVP on 4 processors and class SPVP on 8 or 16 processors.

2.6.2 Strategies for a Dedicated Environment

In a dedicated environment, the strategies for parallel processing are well-understood, and for every problem size at least one unambiguous optimal partition usually exists. The biggest challenge in this environment is developing a low-cost, straightforward partitioning routine.

Problem sizes of VP class should execute on one processor. To preserve single-processor performance for these small problems, first do a quick, inexpensive check to see if the problem to be solved is small. If so, the problem will be solved on one processor regardless of the number of processors requested.

For SPVP and MPVP problems, partition the problem into exactly p subproblems of approximately equal size. The size defined in `NCPUS` will define p. Because it is assumed that exactly p processors will be available, partitioning the problem into fewer than p subproblems ensures an increase in elapsed time, t^e. Partitioning the problem into more than p subproblems degrades vector performance (increasing t^u) for SPVP problems and creates load imbalances for MPVP problems, both causing an increase in t^e.

You cannot partition an LPVP problem into kp subproblems, where k = 2,3,... of equal size in which the subproblems have optimal single-processor (vector) performance. For problems of this size, use a self-scheduling or guided-self-scheduling algorithm to process the kp subproblems and to reduce the effect of losing processors. The possibility of losing a processor in a dedicated environment is small; however, if one is lost, it can dramatically impact performance (doubling t^e) for large problems partitioned into only p subproblems.

2.6.3 Strategies for a Multiuser Environment

In a *multiuser environment*, you are not guaranteed a fixed number of processors. Processors often attach to and detach from jobs unpredictably during execution. Therefore, no optimal time-independent partitioning strategy exists for all problem sizes in this environment. Also, the operating system does not give timely information on how many processors are attached, or may be attached, to your job.

The number of processors that will be attached to your job when entering a parallel region is a function of the number of processors requested, the system load, the time since the last parallel region was encountered, and the settings of a variety of multiprocessing tuning parameters.

Despite this unpredictability, some strategies are available. First, the strategies for VP and LPVP problems are the same as in the dedicated environment. You should execute VP problems on one processor regardless of the number of processors requested, because using one processor is the fastest way to solve the problem.

Secondly, by definition, LPVP problems have optimal vector performance regardless of the number of processors actually attached so that CPU time efficiency e^u should always be close to optimum. LPVP problems are also partitioned into kp subproblems, k = 2,3,..., where p is the requested number of processors. If p' processors ($p'(<)p$) are attached, the ratio (kp/p') is usually large enough that the remainder kp `mod` p' does not create significant load imbalance.

It can be difficult to develop strategies for SPVP and MPVP problems. As in a dedicated environment, you must partition the problem into exactly p subproblems and execute it on p processors. Because the number of processors attached to a job is unknown, however, the correct value of p is not immediately obvious and, unlike the dedicated environment, you should not always set p to `NCPUS`. Choosing p incorrectly could lead to a partitioning strategy that is too aggressive (increasing user and elapsed times) or too conservative (reducing parallelism).

To determine the "best" value of $p \leq$ `NCPUS` (p specifies the number of subproblems and the number of processors to use) for SPVP and MPVP problems, first consider SPVP problems. Let $m = n = 100$ and consider the performance statistics for `SGER`(3S) in Table 1. The Scientific Library subroutine `SGER` is part of Level 2 BLAS (Basic Linear Algebra Subprograms). `SGER` performs the following rank-one update of a general matrix: $A \leftarrow A + axy^T$. A, x, and y are real-valued and of dimension $m \times n$, $m \times 1$, and $n \times 1$, respectively, and a is a scalar.

To process this equation in parallel, partition A horizontally and/or vertically into submatrices A_{ij} with dimensions $m_i \times n_j, i = 1, \ldots n_h;\ j = 1, \ldots n_v;\ n_h, n_u$ are the number of horizontal and vertical partitions, respectively. x and y are partitioned appropriately. This is an SPVP problem because any partition of the original problems produces subproblems that have poorer vector performance.

Table 1. Relative cost for SGER

Number of processors required	Number of processors acquired							
	1	2	3	4	5	6	7	8
1	72.72 (1.00)							
2	84.03 (1.33)	49.79 (0.93)						
3	97.57 (1.80)	70.16 (1.85)	42.63 (1.03)					
4	93.03 (1.63)	54.94 (1.13)	53.21 (1.60)	35.29 (0.94)				
5	126.39 (3.02)	81.47 (2.50)	60.66 (2.08)	58.25 (2.54)	38.23 (1.38)			
6	110.76 (2.32)	64.15 (1.55)	47.88 (1.29)	47.84 (1.70)	45.54 (1.95)	33.56 (1.27)		
7	161.92 (4.95)	98.10 (3.63)	76.81 (3.34)	58.20 (2.54)	57.75 (3.13)	55.70 (3.48)	37.39 (1.84)	
8	127.47 (3.06)	73.02 (2.01)	57.95 (1.90)	45.12 (1.50)	44.01 (1.82)	43.58 (2.15)	41.36 (2.24)	31.72 (1.52)