ParFOR — A Structured Environment for Parallel FORTRAN

by

Wayne Berke

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
715 Broadway, 10th Floor
New York, NY 10003

Ultracomputer Note #137
April, 1988

This work was supported in part by the Applied Mathematical Sciences Program of the US Department of Energy under contract DE-AC02-76ER03077 and in part by the National Science Foundation under grants DCR-8413359, and in part by I.B.M. under joint study agreement N00039-84-R-0605(Q).

ABSTRACT

We present an extension to the FORTRAN language that allows the user to specify parallelism by means of clearly defined, nestable blocks. The implementation achieves compiler-independence through a portable preprocessor. High performance is obtained by *prespawning* processes and relying on a set of run-time routines to manage a self-scheduling allocation scheme. The resulting system, which we call ParFOR, lends itself to the exploitation of fine-grained parallelism because of its low scheduling overhead. It encourages the elimination of explicit process synchronization, thereby enhancing the readability of the source program. In addition, ParFOR provides a variety of modes and compile-time options that are useful for performance measurement and debugging. Finally, we present an evaluation of system efficiency including timing results for several parallel applications running on the eight-processor Ultracomputer prototype.

1. Introduction

It has been shown that parallel programming can be used to provide significant speedups for a variety of applications. A wide range of architectures has been developed that are capable of multiple-CPU concurrency. Unfortunately, this diversity in hardware is also reflected in software. Many parallel programming systems require the user to be intimately acquainted with the architecture. This often results in cryptic, hard-to-maintain code. Clearly, a more general purpose user interface is needed.

One approach is to develop new high-level languages that incorporate parallel semantics within their overall structure. The tasking facilities of Ada are an example of this. Alternatively, more established languages may be enhanced to provide a suitable interface to the multiprocessor. The latter method facilitates the upgrading of existing serial codes without requiring a thoroughgoing translation process. This motivated our decision to extend the syntax of traditional FORTRAN and allow for the explicit specification of parallelism.

Three key design goals have been identified and are discussed below:

- establishing a straightforward syntactic interface
- keeping the implementation reasonably portable
- providing scalable execution speedups proportional to the number of processors

In an effort to shield the programmer from the internals of process management, the syntax is kept as simple as possible and centers upon two kinds of parallel block constructs. For a large class of applications, there is no need to schedule tasks explicitly or do any low level synchronization. All process management is embedded in the run-time routines and thus completely hidden from the programmer.

While initially targeted for the Ultracomputer [Gott 83] [Gott 87], the ParFOR system is also intended for use on other shared-memory, MIMD multiprocessors. Adopting a shared memory model of parallelism allows software to be written for an architecture-insensitive virtual machine. It is more difficult to ‘‘hide the hardware’’ when programming for message-passing multiprocessors. Our concern for portability also motivated a conscious effort to avoid compiler modifications. Consequently, all functionality has been isolated within a set of run-time library routines and a preprocessor that generates calls to them.

The design of the run-time library aided in achieving our final goal of efficiency. ParFOR uses fully dynamic processor allocation to keep processor utilization high, even when contending with nested blocks and with unpredictable execution times of work quanta within a block. Furthermore, we were able to improve performance substantially by confining process management to the user-mode run-time system.

We find the Ultracomputer to be ideally suited as a testbed for systems like ParFOR that rely upon processor self-scheduling. Its *Symunix* operating system [Gott 87], a parallel variant of UNIX†, was designed to take advantage of the *fetch-and-add* primitive in providing a minimum of contention when accessing kernel data structures. Similarly, ParFOR uses fetch-and-add (also referred to here as faa) in assuring that synchronization overhead is minimized as processes contend for work.

2. Related Work

There are several examples of similar efforts at enhancing FORTRAN for multiprocessor execution. Many of them [Prat 87] [Dare 85] are heavily influenced by Jordan's concept of *global parallelism* [Jord 86]. This model effectively turns inside out the traditional means of imposing parallelism upon an essentially single-threaded core. Instead, a program is considered to be intrinsically multi-threaded at the highest scoping level. It then becomes the programmer's responsibility to specify the serial sections by means of barriers and other forms of synchronization.

Many parallel environments impose a special burden upon the programmer to be aware of the parallelism in some way. Dongarra's Schedule system [DonS 86] requires the programmer to specify the dependency relationships among the various modules to be executed. In the PISCES system [Prat 87], the programmer must explicitly describe a virtual machine by mapping it to the ''real'' process model presented by the hardware.

In the ParFOR design, we take the view that the programmer should be isolated from the underlying process model. While semaphores and barriers are available, the programmer is encouraged to rely completely upon the implicit synchronization of the block constructs. Although this may result in a somewhat less general model, the resulting code benefits in terms of both readability and writability. Furthermore, we approximate the effect of global parallelism by prespawning processes that are always available when parallel blocks are entered. In this way, process (and processor) utilization can be very close to optimal.

It should be noted that the PISCES and Schedule systems were designed for distributed as well as shared memory machines. Since ParFOR is not required to adapt itself to the arbitrary process models imposed by message-passing environments, it can more effectively hide the details of the underlying process model from the programmer.

3. Language Features

3.1. Syntax

The programmer may specify parallelism by means of closed block constructs. In this way, the parallel sections of code may be readily identified. Block nesting is fully supported.

Two distinct kinds of parallelism have been identified and we can associate a different block construct with each:[1]

homogeneous (DOALL/ENDALL)

each process executes the same instructions but with some different data values

---

†UNIX is a trademark of AT&T Bell Laboratories.

[1]Analogs of these constructs can be found in [FrJS 85] [Jord 86] [Prat 87].

heterogeneous (PARBEGIN/PARALLEL/PAREND)

each process executes different code sections

3.1.1. Homogeneous parallelism

This mode of parallel execution is closely tied to the uniprocessor notion of iterative loops. Each time around the loop, the program encounters the same instructions but with some different data (in particular, a different index value). Accordingly, the parallel DOALL construct is a straightforward adaptation of the FORTRAN DO. For example:

```
DOALL I=10,300,5
  A(I) = B(I) + B(I-5)
  A(I) = A(I) * 3.14
ENDALL
```

Each iterate of a DOALL block may be scheduled on a different processor if possible. It is the programmer's responsibility to ensure that no data interdependencies exist between the individual iterates. There are currently systems that automate this dependency analysis [AlCK 87], but they are necessarily compiler-based.

3.1.2. Heterogeneous parallelism

When the programmer wishes to specify the parallel execution of a heterogeneous block, an alternative construct may be used. This can be clarified by means of an example:

```
PARBEGIN
   CALL A1
   CALL A2
PARALLEL
   CALL B1
   CALL B2
   CALL B3
PARALLEL
   CALL C1
   CALL C2
PAREND
```

Each pair of statements enclosed by two of the keywords is considered a *subblock* and may be independently scheduled. Thus, the A*n* subroutines may execute in parallel with each B*n* and C*n* as long as the order specified within each subblock is preserved (A1 before A2, etc.).

3.2. The advantages of prespawning

Traditional UNIX systems create and dispose of processes via the *fork* and *exit* system calls, respectively. Symunix provides the *spawn* call which acts as a multi-way fork. In the original ParFOR implementation, a program spawns the necessary *worker processes* at the beginning of each parallel block. These workers then cooperatively execute the individual work units (iterates or subblocks). When the block is finished, each of the workers invokes exit and leaves the system. It was apparent that the overhead for the spawns and exits is expensive, especially when the blocks themselves are relatively small.

An alternative approach was devised that spawned the workers in advance and relied on the run-time system for process management. Each time a parallel block is entered, the prespawned workers are ‘‘woken up’’ and initialized so to begin executing at the first instruction of the block. When the block is finished, the workers invoke a library routine that results in their suspension. This manner of process reuse leads to a substantial performance improvement, especially for programs that display fine-grained parallelism. The current ParFOR system allows the programmer to specify either prespawn or noprespawn mode by means of a command line flag (see **System modes**).

3.3. Semantics of private variables

In most parallel environments, it is useful to be able to consider certain variables as having unique process-specific values. The loop index of a DOALL block is a simple example. We call such variables *private* to distinguish them from shared variables which maintain a single value regardless of the point of reference. On the Ultracomputer, FORTRAN variables are considered private by default. The programmer must explicitly declare shared variables by means of the SHARED declaration.

The process management implied by the previous section suggests conceptualizing worker processes as *stateless* entities. In particular, no assumptions may be made regarding the contents of any private data areas at block entry. This is in contrast to the earlier ParFOR semantics that forced process creation for each block. In that case, programmers could rely on the spawn system call to copy the private segments from the parent to each of its children. In prespawn mode, however, the initial values of private variables are necessarily undefined. For example, consider the following program fragment:

```
SUBROUTINE PAR(MIN)
SHARED /SH/ ARR
INTEGER MIN
INTEGER I, MAX, ARR(20)
MAX = MIN + 100
DOALL I=1,20
   IF (ARR(I).GT.MAX) CALL ERROR
         .
         .
ENDALL
RETURN
END
```

In noprespawn mode, the process executing the serial portion of this code will assign the proper value to the private variable MAX. Upon entering the DOALL block, it will spawn the necessary worker processes which, in turn, inherit that value in their respective copies of MAX. As long as MAX is used read-only within the block, the correctness of the IF condition is assured.

If this program is executed under the prespawn semantics, however, the programmer must be concerned with the stateless property of the workers. Since process creation may have occurred long before the assignment to MAX, the private areas of the workers won’t necessarily reflect that update. If the code executes as written, the IF condition may be evaluated with erroneous results.

3.4. The COPYIN statement

Observe that the above code may be made to work in prespawn mode by simply copying MAX to a shared variable before the DOALL and then substituting the copy within the IF condition. In general, this can be done for any private variable that needs an initial value and is used read-only within the block. For a more general facility and to ease the serial to parallel conversions of existing codes, we have supplemented our enhanced FORTRAN with the COPYIN statement. This statement declares the private variables that are to be copied from the process initiating the parallel block. We can thus add the statement

```
COPYIN MAX
```

after the DOALL statement to ensure that each worker obtains an initialized copy of that variable.

We see that the programmer must bear the burden of detecting such variables. In general, they include any private variables that can be read before they are written in any iterate or subblock. They may be local to a procedure, part of a common block, or a subroutine parameter aliasing either of these. If subroutines are called from within parallel sections, the former must also be examined.

It would seem that certain data flow analysis techniques could be used to find variables that may have undefined values because they weren't copied in. Similar techniques to those for detecting data-dependencies could be brought to bear in this effort. However, providing such a facility would necessarily have to be built into the compiler and would thus render the system less portable.

4. Implementation overview

4.1. Preprocessor

The preprocessor acts as a single-pass text filter that converts a ParFOR program into FORTRAN 77. Only the special parallel constructs are transformed, the rest of the program remains as before. The transformations are straightforward and much of the complexity is hidden within the run-time library routines. For example, if we applied the preprocessor to our earlier DOALL example, we have the following result:

```
      logical getix
      call doall(1,I,10,300,5,0,0,0)
89701 if (getix()) then
         A(I) = B(I) + B(I-5)
         A(I) = A(I) * 3.14
         go to 89701
      endif
```

Thus the DOALL block is transformed into the FORTRAN equivalent of a WHILE loop. The entire interface to the run-time system consists of the calls to doall[2] and getix. The former is invoked once before loop entry and

[2]Any preprocessor that introduces new symbol names or label values runs the risk of clashing with programmer-defined names or labels. We deal with this by providing two versions of ParFOR. In the system as presented above, the preprocessor considers certain label values to be within its ''reserved'' range. It is up to the programmer not to use the names doall or `getix` for any program objects. The other ''safer'' version actually disguises the labels and function names by embedding the character '%' within them. However, this implementation also requires some modification to the lexical front ends of the compiler and loader. The safer version is the one currently running on the Ultracomputer.

sets up the data structure for this particular parallel block activation. The getix routine is called before each loop iterate.

In a similar way, the integer function getpb is invoked once for each subblock of a heterogeneous block. Applying the preprocessor to our earlier example, we generate:

```
      integer getpb
      go to 89712
89713     continue
            CALL A1
            CALL A2
          go to 89711
89714     continue
            CALL B1
            CALL B2
            CALL B3
          go to 89711
89715     continue
            CALL C1
            CALL C2
          go to 89711
89712     call doall(1,0,1,3,1,0,0,0)
89711     go to (89713,89714,89715,89716) getpb()
89716 continue
```

4.2. Run-time system

As mentioned earlier, all dynamic resource management is handled by the run-time system. There is a shared memory allocator that provides support for the multiple execution environments that exist when parallel blocks are nested. We implement this by pre-allocating an arena of memory to be managed by the run-time unit.

There are three entry points to the run-time system: the routines doall, getix, and getpb. The doall call serves to set up the environment for parallel execution. First, a shared *work control block* (wcb) is allocated and initialized according to the parameters in the call. All necessary information about a given parallel block is maintained here and referenced within subsequent calls to the run-time system. Second, multiple worker processes are activated[3] and begin executing at the instruction after the call to doall, i.e. at the call to getix or `getbp`. The main function of the wcb is to act as a communications area among all processes assigned to a particular block.

The getix routine is used in the implementation of homogeneous blocks. Its function is to obtain a unique value for the loop index within the range specified by the DOALL statement. It does this by performing a *fetch-and-add*[4] operation on the *iteration generator* (located within the wcb) for the particular block. The faa return value is then assigned to the loop index of the block, provided it is within the DOALL range. When a worker obtains a

[3]If noprespawn mode has been specified, the worker processes must be created first.

[4]The fetch-and-add operation is the basic synchronization primitive of Ultracomputer-like machines. The operation *faa(v, e)* atomically adds the expression *e* to the variable *v* and returns the original value of *v*. Thus, if concurrent processes call *faa* with the same (non-zero) arguments, each is assured of a unique return value. Furthermore, when different processes issue faa's to the the same variable, the multiple operations may be *combined* by hardware within the interconnection network. This permits the implementation of many different flavors of process synchronization without inducing critical sections [Gott 87].

value that is out of range, no assignment is made and the process suspends itself or exits. Similarly, when the parent determines that the DOALL range has been exhausted, a FALSE return from getix causes the program to branch over the loop.

In the case of heterogeneous blocks, the iteration generator is used in a slightly different way. Each subblock is set up as the target of a computed GOTO so that the worker processes can use the faa return (obtained through getpb) to guide them to their own unique portion of the code. Block termination occurs when getpb return an integer that is one greater than the total number of subblocks.

4.3. Process management

All process allocation is done at run-time on a demand-driven basis. This is in contrast to other methods that rely upon statically allocating processes to groups of work units. The advantage of dynamic allocation is evident when dealing with a block whose individual units may vary in execution time among each other. Even in a homogeneous block, a process may be statically assigned a fixed number of iterates that happen to complete very quickly because of conditional execution within the loop. Another process may be assigned the same number of iterates but which require substantially more execution time. The first process will have to idle while the other plods along, sequentially finishing its allotted work. The fully dynamic semantics of ParFOR ensure that if there are work units left undone, and there are worker processes that are idle, the workers will be dispatched to the unfinished work units. Moreover, this property holds even when parallel blocks are nested.

Proponents of static scheduling point to the fact that by pre-allocating groups of work units (*static chunking*), they minimize scheduling overhead. Furthermore, many numerical applications are dominated by matrix computations and can be partitioned into homogeneous iterates with comparable execution times. This reasoning becomes problematic in the context of nested parallel blocks, however. In particular, when a subroutine containing a DOALL is called from within another parallel block, we cannot rely on static chunking to provide maximum processor utilization.

Jordan claims that self-scheduling imposes a minimum size criteria for parallel blocks in order to amortize the cost of scheduling each iterate [Jord 86]. However, ParFOR limits this cost to only a few instructions. Of this number, there is but a single shared memory access which fetch-and-add's the iteration generator. If fetch-and-adds can be combined, this access does not require any serialization of processes working on the same parallel block. Consequently, the overhead of scheduling an iterate is comparable to that of a simple subroutine call.

The self-service paradigm of process allocation centers around a concurrent-read exclusive-write linked list. When a process initiates a parallel block, it instantiates a wcb and inserts it onto the globally shared work list. Worker processes poll the list to determine when it becomes non-empty. When the work becomes available, the competing processes each try to read the last wcb on the list.

Access to the list is arbitrated via a readers/writers lock. There are four operations associated with the list:

llremove
: removes a node (work control block) after first locking as a writer

llput
: inserts a node after locking as a writer

llget
: reads the last element on the list after locking as a reader

llempty
: tries to read the last element without any prior locking and returns FALSE if successful (TRUE otherwise)

The body of the suspend routine is then (expressed in PASCAL):

```
REPEAT
   REPEAT
      "busy wait in cache a few microseconds"
   UNTIL NOT llempty;
   curdoall := llget
UNTIL curdoall <> NIL;
dispatch (curdoall);
```

In this way, we minimize accesses to shared memory and avoid locking the list while polling. When we have evidence of list occupancy, the list is locked as a reader and accessed via llget. Note that the list may be locked as a writer while other processes are testing it via llempty but not during `llget`. Because of this, it is possible for the list to be nonempty during the former operation but empty during the latter. Such a scenario does not lead to difficulties, however, as the outer REPEAT simply begins another iterate.

5. Portability

ParFOR is designed for a shared-memory MIMD multiprocessor with a UNIX-based operating system. The preprocessor is written completely in C as is the bulk of the run-time library. A 50-line assembler file (the current prototype is 68010-based) is required to implement low-level process dispatching. The system is remarkably compact: there are about 1,500 source lines for the preprocessor and 1,000 for the run-time system.

Much of the work in porting the system should be confined to interfacing the run-time routines with the process and memory management primitives of the target operating system. It should been emphasized that ParFOR only invokes these operating system calls once at the time of the prespawn. After that, resource management becomes the sole province of the user-mode run-time routines.

6. System Modes

We allow the programmer to specify the particular mode by means of a command line flag. Valid modes are:

prespawn **(-Ppre)**

Processes are created once only and assigned by demand to subtasks.

noprespawn **(-Pnopre)**

Worker processes must be created anew at each new block invocation (within the constraints of the specified degree of parallelism). When the block terminates, they are destroyed.

serial **(-Pser)**

This mode converts a program written in parallel FORTRAN to be converted to its serial equivalent. All DOALL blocks are converted to DOs. Heterogeneous blocks simply execute the PARALLEL subblocks one after the other. In addition, shared variables are converted to normal common block globals (this allows them to be considered cacheable).

The different modes allow the user to exploit different facilities in the parallel environment without altering the original source code. Serial mode is particularly useful in comparing the execution times of a parallel program and its serial equivalent.

7. Additional Features

7.1. Selective deparallelization

As a generalization of serial mode, we allow the programmer to turn off the parallelism of individual blocks. This can be done by simply adding the character '*' after the block entry keyword (DOALL or PARBEGIN). This facility, in combination with the different system modes, can aid the user in tracking down a program error related to process interaction.

7.2. Varying the degree of parallelism

By default, the system provides a parallel program with as many processes as there are processors. This limit may be specified differently by two means:

- If the application is compiled under prespawn mode, an explicit PRESPAWN statement can specify the maximum parallelism.
- The shell environment variable DOALLPES can be set to the desired level of parallelism. This allows the programmer to test an application with different numbers of processes, but without recompiling the program.

In either method, the specified degree must be a positive integer no greater than the number of physical processors.

7.3. Debugging

A simple debugging strategy is to compile the program into three executables by using the different options; let **A** be the result of serial mode, **B** the result of noprespawn mode, and **C** the result of prespawn mode. If **A** and **B** execute correctly, but **C** does not, the likely culprit is a renegade private variable that should have been copied in. If **A** executes correctly, but **B** and **C** do not, the problem is linked to parallelism but its exact nature is not yet clear. If all three execute incorrectly, the bug has nothing to do with parallelism and the standard tools for debugging serial programs can be used.

In the first two cases, where the bug is somehow related to parallelism, the preprocessor can be used to pinpoint the specific block that is responsible for the anomalous behavior. This can be done by selectively deparallelizing each block until the program begins to execute correctly. If there are a large number of parallel blocks in the program, half of them can be turned off and the user can proceed in a kind of binary search until the problem block is found. Note that this technique is most useful when programs exploit finer granularities of parallelism.

8. Performance measurement

8.1. Expected speedups

An important property of any parallel software system is its ability to provide programmer access to multiple processors without introducing significant performance delays. This can be measured by comparing a parallel execution time to its application-equivalent serial time. For a given parallel block, we can express this relation as its *speedup ratio*:

$$\text{speedup} = \frac{\text{serial time}}{\text{parallel time}} = \frac{wn_w}{O_b + \dfrac{n_w(O_w + w)}{\min(p, n_w)}}$$

where

O_b	is the overhead to start up the parallel block
O_w	is the overhead to schedule a work unit
n_w	is the number of work units in the block
w	is the average size of each work unit in the block
p	is the number of processors

Note that, for simplicity, our model assumes negligible memory contention among concurrent work units. If $n_w \geq p$ above, the equation simplifies to

$$\frac{1}{\frac{O_b}{wn_w} + \frac{O_w}{wp} + \frac{1}{p}}$$

It is clear from the above expression that optimal system performance occurs when the two overheads are zero, thus rendering speedup equal to the number of processors. In a more realistic context of non-zero overheads, we can amortize their cost by maximizing w and n_w, thereby obtaining a speedup close to p.

ParFOR's prespawn mode values for O_b and O_w come to approximately 160 and 10 machine instructions, respectively. This supports our claim that blocks with small individual work units can exhibit good speedups if there are enough of them. For example, in a 64-processor configuration, we can achieve a speedup of 50 when $w = 50$ and $n_w = 5000$. A speedup of 60 is obtained if $w = 200$ and $n_w = 10000$. Note that these values for w are in units of machine language instructions.

8.2. Timing results

There were a number of FORTRAN applications that were useful in providing testing grounds for the new system. The programs tested were:

peskin	Charles Peskin's more simple model of blood flow through the heart valves
rantst	a concurrent test program for a new random number generator
femtst	a linear system solver that uses finite element methods
simple	solution of partial differential equation for hydro-dynamics and heat conduction

By using the various ParFOR options, we were able to get an idea of how close to ideal speedups we can expect under realistic circumstances. A single source for each application was preprocessed and compiled under the three modes. The results:

		serial	noprespawn		prespawn	
		time	time	*speedup*	time	*speedup*
peskin	total	10960	4854	*2.3*	2215	*4.9*
	selected	10237	4124	*2.5*	1480	*6.9*
rantst	total	5788	3430	*1.7*	1023	*5.7*
	selected	5568	3252	*1.7*	801	*7.0*
femtst	total	826	485	*1.7*	183	*4.6*
	selected	821	454	*1.8*	146	*5.6*
simple	total	177	80	*2.2*	59	*3.0*
	selected	147	47	*3.1*	25	*5.9*

All measurements were taken on a lightly loaded machine operating in multiuser mode. Individual times are given in seconds with speedups vs. serial in italics. The timing values reflect actual elapsed (‘‘wall clock’’) time with a precision of 1/16 second. In addition to timing the entire application, the *selected* measurements show the time spent within the parallelized blocks only. All 8 processors were used for prespawn and noprespawn modes.

Several factors combined to keep the parallelized versions from realizing maximum speedup. As noted in the previous section, the greater the size of the parallel block or the more numerous the number of work units within it, the less the effect on overall performance by the run-time routine overheads. This is borne out by the observation that the best speedups occur in **peskin** and **rantst**, both of which contain DOALL’s with sizable loop bodies. The hardware factors that limit performance derive from the presence of a write-through cache and a shared bus to global memory. This results in the serialization of all non-cache memory accesses as well as all writes to memory, whether in-cache or not. The replacement of the shared bus with an omega network in a future Ultracomputer prototype will increase the bandwidth to global memory and should push these speedups even closer to the optimal.

9. Conclusions

We have successfully designed and implemented a parallel FORTRAN language extension and execution environment. Its efficiency and the relative simplicity of its user interface have already made it popular among programmers. It may be ported to any shared memory multiprocessor system and is especially suited to those running under UNIX-like operating systems. We have achieved excellent speedups for a variety of applications by prespawning worker processes and relying on the user-mode library to perform dynamic process allocation. Since the

programmer is no longer concerned about the scheduling overhead of small blocks, programming styles featuring fine-grained parallelism should be encouraged. Fine-grained parallelism, in conjunction with an assortment of debugging modes, should enhance the user's ability to track down a difficult bug by locating the specific block that causes it.

References

[AlCK 87]
J. R. Allen, D. Callahan, and Ken Kennedy, ‘‘Automatic Decomposition of Scientific Programs for Parallel Execution’’, *Conference Record 14th Annual Symposium of the Principles of Programming Languages*, January, 1987, pp. 63-76.

[Cray 86]
Cray Computer Systems, *Multitasking User Guide*, Technical Note SN-0222, (Chapter 12 – Micro-tasking), 1986.

[Dare 85]
Frederica Darema-Rogers, et. al., ‘‘A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN’’, IBM Research RC11552, 1986.

[DonS 86]
Jack Dongara and Danny Sorensen, ‘‘SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs’’, Technical Report ANL/MCS-TM-86, Argonne National Laboratory, University of Chicago, 1986.

[FrJS 85]
Paul Frederickson, Rondall Jones, and Brian Smith, ‘‘Synchronization and control of parallel algorithms’’, *Parallel Computing*, November, 1985, pp. 255-264.

[Gott 83]
Allan Gottlieb, et. al., ‘‘The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer’’, *IEEE Trans. Comp.*, pp. 175-189, February 1983.

[Gott 87]
Allan Gottlieb, ‘‘An Overview of the NYU Ultracomputer Project’’, Ultracomputer Note 100, Courant Institute, NYU, 1987 (to appear in *Experimental Parallel Computing Architectures*, Jack Dongarra, ed.).

[Jord 86]
Harry F. Jordan, ‘‘Structuring parallel algorithms in an MIMD, shared memory environment’’, *Parallel Computing*, May, 1986, pp. 93-110.

[Prat 87]
Terrence Pratt, ‘‘The Pisces 2 Parallel Programming Environment’’, *International Conference on Parallel Processing*, 1987, pp. 439-445.