

# VAST/Parallel: Automatic Parallelization for SMP systems

Chip Rodden and Brian Brode  
Pacific-Sierra Research Corporation  
2901 28<sup>th</sup> Street, Santa Monica, CA 90405  
info@psrv.com

## Abstract

VAST-C/Parallel (C) and VAST-2/Parallel (Fortran) are automatic parallelizers for SMP systems. They analyze existing programs and transform them into parallel programs that use threads to process calculations (such as loop nests) concurrently. This paper examines their features and functions.

## 1 Introduction

The VAST/Parallel products are automatic parallelizing preprocessors that can improve the performance of Fortran and C applications on shared memory parallel (SMP) computer systems. They allow existing codes to use the multiple processors of an SMP system by automatically restructuring the input source to use parallel threads.

The basic operation of VAST/Parallel is to identify parallel regions in the source code, and then create and call new parallel routines that contain these parallel regions in place of the original code. The parallel routines are invoked on each thread, with each thread getting a different portion of the computation. Between execution of parallel regions, execution is on one (master) thread.

## 2 Parallel Loop Nests

The basic unit of analysis for the VAST/Parallel system is the loop nest. Loops are analyzed in simple and complicated loop nests; loops containing the largest amount of work are parallelized where possible. Loops do not have to be tightly nested. In Fortran 90 code, VAST-2/Parallel can in general parallelize and optimize multi-dimensional array syntax just as efficiently as loop nests. Array syntax is in fact transformed into optimized fused nested loops that are then partitioned across the parallel processors.

Using extensive data dependency analysis, all loop nests are examined to see if they are safe for parallel execution. VAST/Parallel can resolve ambiguous subscripting by examining variable assignments outside of loops, and restructure the use of variables to avoid certain other dependencies. When dependencies are unclear at compile time, sometimes VAST/Parallel can generate run-time tests to allow parallelism to proceed. (With C, it is often helpful for the user to specify assertion levels that help disambiguate pointers that may be passed to functions. This is easy to do and can make a big difference depending on how the C code is written.)

Summations and other reductions are parallelized through the use of locks or critical regions. Each thread calculates its contribution to the summation in parallel, and then all of the contributions are combined one at a time in a locked region to get the final sum.

When using VAST/Parallel, there is a choice of static or dynamic partitioning of loop iterations. Static partitioning gives each available thread an equal number of the total iterations, and each thread processes its piece. Dynamic partitioning gives each thread a chunk of iterations, then when the thread is done with that chunk, it checks to see if there are more chunks to be done, and if so it gets the next chunk. Static partitioning is good because it has very low overhead – the iterations that each thread should calculate are quick to compute. Dynamic partitioning is good when loop iterations can have very different execution times (due to conditionally executed code, for instance). The computational load may be better balanced

by allowing threads to dynamically accept smaller chunks of work. The user can switch between these methods, and use dynamic partitioning for loop nests that need more load balancing, and static partitioning when low overhead is more important. The default for most loops is static partitioning.

### **3 Parallel Sections**

In addition to loop nest analysis, VAST/Parallel has automatic recognition of parallel sections. When pieces of code deal with disjoint operations, VAST/Parallel can process each piece in a separate parallel section, with a different thread executing each disjoint piece of code.

Using interprocedural analysis, VAST/Parallel can examine call chains to determine their dependencies, and then parallelize loops containing calls or groups of calls outside loops. These groups of calls are processed in parallel sections, with each disjoint call processed by a different thread.

### **4 Parallel Regions**

All variables in a parallel region are categorized as shared (seen by all threads) or private (local copy in each thread). VAST/Parallel can detect and create private arrays.

VAST/Parallel extends parallel regions to include multiple parallel loop nests, non-parallel outer loops, and intervening scalar code. This cuts down on parallel overhead at the cost of some redundant execution.

When VAST/Parallel finds a parallel region, if the amount of work in the region is not clear at compile time, then VAST/Parallel creates a run-time threshold test. Through this run-time test, the parallel region will only be executed if there is enough work; otherwise, the original serial version is executed. This lets only the most computationally intensive loops execute in parallel, and keeps tiny loops in scalar.

VAST/Parallel includes high-level scalar optimizations as well as automatic parallelization, to help your performance even in a single thread. Parallel optimizations can be done to outer loops while inner loops are optimized for efficient execution on one thread.

The number of threads can be set with an environment variable. This allows the degree of parallelism to be changed from run to run. When the system is busy you can run with two threads, when it is empty you can run with eight threads, without recompiling your program.

### **5 Using VAST/Parallel**

VAST/Parallel products are automatic tools which can be used by both knowledgeable and neophyte users. A less knowledgeable person can run the VAST/Parallel product on his code with default options and potentially get some increase in performance. A sophisticated user can use the many options and directives to customize the way VAST/Parallel optimizes a particular program.

The VAST/Parallel products come with driver programs to make their use as direct as the non-parallel native compiler. The drivers allow the passing of compiler and VAST/Parallel options and automatically link in the necessary libraries to run a threaded program.

Here is a simple example of how one might go about parallelizing one's program and the mechanism the VAST/Parallel products use to gain a performance increase. To run VAST/Parallel on a Fortran program in source file program.f on a Unix system, you can use the following command (pfor stands for "parallel fortran"):

```
pfor program.f
```

This will create an executable which can then be run on the SMP machine in parallel. Any other switches and options you would use for your program on a non-parallel compile would be put instead on the pfor command.

## 6 Example Code

The actual code generated by VAST/Parallel is normally passed to the native compiler on the target system, and the user does not look at it. However, to get a better understanding of what is actually happening, let's examine the transformations done by VAST/Parallel to a simple program by looking at the intermediate code. Examine the following routine:

```

subroutine exml ( c )
real  a(200,2000),b(200,2000),c(200,2000)

do j = 1,1000
  do i = 1,100
    c(i,j) = b(i,j) * a(i,j)
  enddo
enddo

end

```

VAST-2/Parallel would transform this into the following:

```

subroutine exml ( c )
C...Translated by Pacific-Sierra Research VAST-2  6.0I4
C...Switches: -pl
real  a(200,2000),b(200,2000),c(200,2000)

external _vp_exml1
integer j3, j4, j5
real r1, r2, r3, r4

call vp_pcall (_vp_exml1 , 2, 3, b, a, c)

end

subroutine _vp_exml1 (vp_mythread, vp_nmthreads, b, a, c)
REAL b(200,2000), a(200,2000), c(200,2000)
INTEGER j, i, j3, j4, j5, vp_mythread, vp_nmthreads, j1, j2
REAL r1, r2, r3, r4

j3 = (250 + vp_nmthreads - 1)/vp_nmthreads
j4 = vp_mythread*j3 + 1
j5 = min(250 - j4 + 1, j3)
do j = j4, j5 + j4 - 1
  do i = 1, 100
    r1 = b(i,j*4-3)*a(i,j*4-3)
    r2 = b(i,j*4-2)*a(i,j*4-2)
    r3 = b(i,j*4-1)*a(i,j*4-1)
    r4 = b(i,j*4)*a(i,j*4)
    c(i,j*4-3) = r1
    c(i,j*4-2) = r2
    c(i,j*4-1) = r3
    c(i,j*4) = r4
  end do
end do

```

```
return
end
```

You'll notice first that the parallelized code is "split" into a separate routine "\_vp\_exml1". This routine is then called as input to "vp\_pcall" which initiates the parallelism. The parallel code is statically segmented across each of the processors inside the parallel routine by basically dividing the parallel loop ( the j loop in this example ) into the "number of iterations/number of threads". So if the number of threads is two then each processor would receive half the work of the j loop.

The interface to the "vp\_pcall" routine is:

First argument:	Parallel routine
2nd argument:	Number of threads (default 2, environment variable NTHREADS)
3rd argument:	Number of "shared" variables
4th-Nth arguments:	Shared variables

"Shared" variables are those which are "global" to all processors and need to be changed and/or used by each processor. All other variables in a parallel routine are "private" variables, which will be declared locally to the routine. These variables will be on separate stacks in each thread.

The VAST/Parallel paradigm is to start up the threads in the first call to vp\_pcall and then have them spin-wait while they wait for work to do. This greatly decreases overhead of continually creating/killing the threads every time there is parallel work to be done. It does, however, assume a dedicated system and because many parallel machines are multi-user, VAST/Parallel has the option of having the threads sleep-wait, where they are explicitly put to sleep until they are woken up for the next parallel region.

You'll also notice that a scalar optimization of loop unrolling is done on the loop as well. This can help the compiler optimize the loop, creating even greater potential speed-up.

VAST-C/Parallel translation of C programs is similar in concept and capabilities to this Fortran example.

## 7 Benchmark Timings

Below are some benchmark timings done on a 2-headed RS6000 workstation. They were done on a dedicated machine using static partitioning. They were run with inlining (-e78) and by parallelizing inner loops (-ei) and -O to the native compiler.

<i>Benchmark</i>	<i>Speedup</i>
<b>tomcatv</b>	<b>1.6</b>
<b>hydro2d</b>	<b>1.8</b>
<b>dnasa7</b>	<b>1.5</b>
<b>matrix300</b>	<b>3.7</b>
<b>lss</b>	<b>1.6</b>

As you can see, significant speed-ups can be obtained with very little work on the part of the user. Of course, these benchmarks are generally extremely parallel and speed-ups are greatly influenced by the coding style of the user. However, VAST/Parallel can help restructure your code, if necessary, by giving informative messages as to why each loop was not optimized (it will produce an annotated listing file). These problems are sometimes just "potential" ones that the user can "override" with directives or options thereby giving him increased performance.

## 8 User-Directed Parallelism

For situations where automatic parallelization is not sufficient, the user can take advantage of the built-in user-directed parallelism directives. These allow the user to explicitly denote the parallel regions, parallel loops, parallel sections, shared/private variables, and reduction operations in the program. With these directives, the user bypasses VAST/Parallel's automatic parallelization analysis and uses VAST for the mechanics and bookkeeping of creating and invoking the parallel routines.

## **9 Final Comments**

VAST/Parallel products are available on these systems:

IBM RS/6000 (AIX)

Intel Pentium (Windows NT, LINUX)

Sun SPARC (Solaris)

DEC Alpha (Unix)

SGI (IRIX)

VAST/Parallel can offer both the knowledgeable and neophyte user significant increases in their program's speed on parallel systems. It can make it very easy for users to adapt their code to run on SMP machines.