OpenMP Tutorial
Part 2: Advanced OpenMP

Rudolf Eigenmann
Purdue University
School of Electrical and
Computer Engineering

Tim Mattson
Intel Corporation
Computational Software Laboratory

SC’ 2000 Tutorial Agenda

- ➡ Summary of OpenMP basics
- OpenMP: The more subtle/advanced stuff
- OpenMP case studies
- Automatic parallelism and tools support
- Mixing OpenMP and MPI
- The future of OpenMP

Summary of OpenMP Basics

- Parallel Region

C$omp parallel	#pragma omp parallel

- Worksharing

C$omp do	#pragma omp for
C$omp sections	#pragma omp sections
C$omp single	#pragma omp single
C$omp workshare	#pragma omp workshare

- Data Environment
  - ◆ directive: threadprivate
  - ◆ clauses: shared, private, lastprivate, reduction, copyin, copyprivate
- Synchronization
  - ◆ directives: critical, barrier, atomic, flush, ordered, master
- Runtime functions/environment variables

Agenda

- Summary of OpenMP basics
- ➡ OpenMP: The more subtle/advanced stuff
  - ◆More on Parallel Regions
  - ◆Advanced Synchronization
  - ◆Remaining Subtle Details
- OpenMP case studies
- Automatic parallelism and tools support
- Mixing OpenMP and MPI
- The future of OpenMP

OpenMP: Some subtle details

- Dynamic mode (the default mode):
  - The number of threads used in a parallel region can vary from one parallel region to another.
  - Setting the number of threads only sets the maximum number of threads - you could get less.
- Static mode:
  - The number of threads is fixed between parallel regions.
- OpenMP lets you nest parallel regions, but…
  - A compiler can choose to *serialize* the nested parallel region (i.e. use a team with only one thread).

Static vs dynamic mode

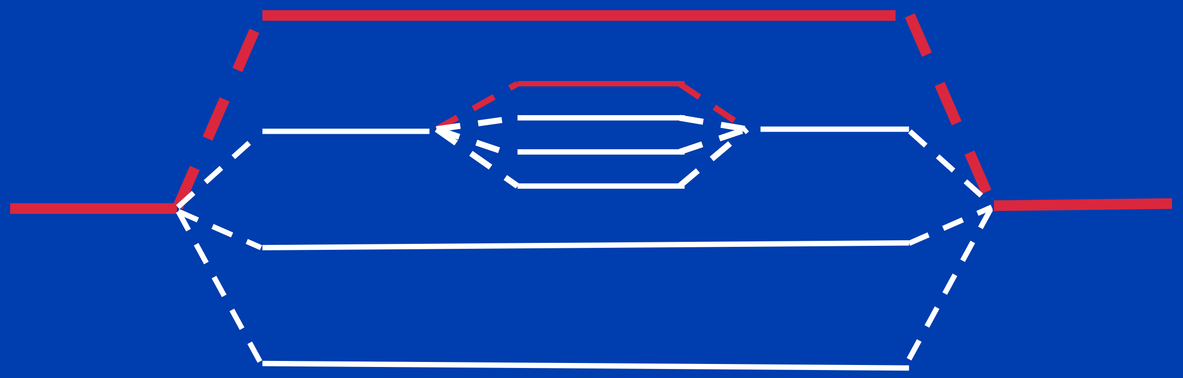
- An example showing a static code that uses threadprivate data between parallel regions.

EPCC Microbenchmarks

- A few slides showing overheads measured with the EPCC microbenchmarks.

Nested Parallelism

- OpenMP lets you nest parallel regions.



- But a conforming implementation can ignore the nesting by serializing inner parallel regions.

OpenMP:
The numthreads() clause

New in OpenMP 2.0

- The numthreads clause is used to request a number of threads for a parallel region:

Any integer expression

```
           integer id, N
C$OMP PARALLEL NUMTHREADS(2 * NUM_PROCS)
                id = omp_get_thread_num()
                res(id) = big_job(id)
C$OMP END PARALLEL
```

- NUMTHREADS only effects the parallel region on which it appears.

Nested parallelism challenges

- Is nesting important enough for us to worry about?
- Nesting is incomplete in OpenMP. Algorithm designers want systems to give us nesting when we ask for it.
    - What does it mean to ask for more threads than processors? What should a system do when this happens?
- The set_num_threads routine can only be called in a serial region. Do all the nested parallel regions have to have the same number of threads?

OpenMP:
The if clause

- **The if clause is used to turn parallelism on or off in a program:**

Make a copy of id for each thread.

```
        integer id, N
C$OMP PARALLEL PRIVATE(id)  IF(N.gt.1000)
            id = omp_get_thread_num()
            res(id) = big_job(id)
C$OMP END PARALLEL
```

- **The parallel region is executed with multiple threads only if the logical expression in the IF clause is .TRUE.**

OpenMP:
OpenMP macro

- OpenMP defines the macro _OPENMP as **YYYYMM** where YYYY is the year and MM is the month of the OpenMP specification used by the compiler

```
        int id = 0;
#ifdef _OPENMP
            id = omp_get_thread_num();
            printf(“ I am %d \n”,id);
#endif
```

OpenMP: Environment Variables: The full set

- Control how “omp for schedule(RUNTIME)” loop iterations are scheduled.
  - OMP_SCHEDULE “schedule[, chunk_size]”
- Set the default number of threads to use.
  - OMP_NUM_THREADS *int_literal*
- Can the program use a different number of threads in each parallel region?
  - OMP_DYNAMIC TRUE || FALSE
- Do you want nested parallel regions to create new teams of threads, or do you want them to be serialized?
  - OMP_NESTED TRUE || FALSE

OpenMP: Library routines: Part 2

- Runtime environment routines:
  - Modify/Check the number of threads
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Turn on/off nesting and dynamic mode
    - omp_set_nested(), omp_get_nested(), omp_set_dynamic(), omp_get_dynamic()
  - Are we in a parallel region?
    - omp_in_parallel()
  - How many processors in the system?
    - omp_num_procs()

Agenda

- Summary of OpenMP basics
- OpenMP: The more subtle/advanced stuff
  - More on Parallel Regions
  - ➡ Advanced Synchronization
  - Remaining Subtle Details
- OpenMP case studies
- Automatic parallelism and tools support
- Mixing OpenMP and MPI
- The future of OpenMP

OpenMP: Library routines: The full set

- Lock routines
  - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock()

… and likewise for nestable locks

- Runtime environment routines:
  - Modify/Check the number of threads
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Turn on/off nesting and dynamic mode
    - omp_set_nested(), omp_get_nested(), omp_set_dynamic(), omp_get_dynamic()
  - Are we in a parallel region?
    - omp_in_parallel()
  - How many processors in the system?
    - omp_num_procs()

OpenMP: Library Routines

- **Protect resources with locks.**

```
    omp_lock_t lck;
    omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
    id = omp_get_thread_num();
    tmp = do_lots_of_work(id);
    omp_set_lock(&lck);
      printf(“%d %d”, id, tmp);
    omp_unset_lock(&lck);
}
```

Wait here for your turn.

Release the lock so the next thread gets a turn.

OpenMP: Atomic Synchronization

- Atomic applies only to the update of x.

```
C$OMP PARALLEL PRIVATE(B)
          B =  DOIT(I)
C$OMP ATOMIC
          X = X + foo(B)

C$OMP END PARALLEL
```

```
C$OMP PARALLEL PRIVATE(B, tmp)
          B =  DOIT(I)
          tmp = foo(B)
C$OMP CRITICAL
          X = X + tmp

C$OMP END PARALLEL
```

Some thing the two of these are the same, but they aren’t if there are side effects in foo() and they involve shared data.

OpenMP: Synchronization

- The flush construct denotes a sequence point where a thread tries to create a consistent view of memory.
  - All memory operations (both reads and writes) defined prior to the sequence point must complete.
  - All memory operations (both reads and writes) defined after the sequence point must follow the flush.
  - Variables in registers or write buffers must be updated in memory.
- Arguments to flush specify which variables are flushed. No arguments specifies that all thread visible variables are flushed.

OpenMP:
A flush example

- This example shows how **flush** is used to implement pair-wise synchronization.

```
          integer ISYNC(NUM_THREADS)
C$OMP PARALLEL DEFAULT (PRIVATE) SHARED (ISYNC)
          IAM = OMP_GET_THREAD_NUM()
          ISYNC(IAM) = 0
C$OMP BARRIER
          CALL WORK()
          ISYNC(IAM) = 1   ! I'm all done; signal this to other threads
C$OMP FLUSH(ISYNC)
          DO WHILE (ISYNC(NEIGH) .EQ. 0)
C$OMP FLUSH(ISYNC)
          END DO
C$OMP END PARALLEL
```

Make sure other threads can see my write.

Make sure the read picks up a good copy from memory.

Note: OpenMP's flush is analogous to a fence in other shared memory API's.

OpenMP:

Implicit synchronization

- Barriers are implied on the following OpenMP constructs:

```
end parallel
end do  (except when nowait is used)
end sections (except when nowait is used)
end single (except when nowait is used)
```

- Flush is implied on the following OpenMP constructs:

```
barrier
critical, end critical
end do
end parallel
```

```
end sections
end single
ordered, end ordered
parallel
```

Synchronization challenges

- OpenMP only includes synchronization directives that “have a sequential reading”. Is that enough?
  - Do we need conditions variables?
  - Monotonic flags?
  - Other pairwise synchronization?
- When can a programmer know they need or don’t need flush? If we implied flush on locks, would we even need this confusing construct?

Agenda

- Summary of OpenMP basics
- OpenMP: The more subtle/advanced stuff
  - ◆More on Parallel Regions
  - ◆Advanced Synchronization
  - ➡ ◆Remaining Subtle Details
- OpenMP case studies
- Automatic parallelism and tools support
- Mixing OpenMP and MPI
- The future of OpenMP

OpenMP:
Some Data Scope clause details

- The data scope clauses take a list argument
  - The list can include a common block name as a short hand notation for listing all the variables in the common block.
- Default private for some loop indices:
  - Fortran: loop indices are private even if they are specified as shared.
  - C: Loop indices on “work-shared loops” are private when they otherwise would be shared.
- Not all privates are undefined
  - Allocatable arrays in Fortran
  - Class type (I.e. non-POD) variables in C++.

See the OpenMP spec. for more details.

OpenMP: More subtle details

- Variables privitized in a parallel region can not be reprivitized on an enclosed omp for.
- Assumed size and assumed shape arrays can not be privitized.
- Fortran pointers or allocatable arrays can not lastprivate or firstprivate.
- When a common block is listed in a data clause, its constituent elements can’t appear in other data clauses.
- If a common block element is privitized, it is no longer associated with the common block.

This restriction will be dropped in OpenMP 2.0

OpenMP:
directive nesting

- For, sections and single directives binding to the same parallel region can’t be nested.
- Critical sections with the same name can’t be nested.
- For, sections, and single can not appear in the dynamic extent of critical, ordered or master.
- Barrier can not appear in the dynamic extent of for, ordered, sections, single., master or critical
- Master can not appear in the dynamic extent of for, sections and single.
- Ordered are not allowed inside critical
- Any directives legal inside a parallel region are also legal outside a parallel region in which case they are treated as part of a team of size one.

Agenda

- Summary of OpenMP basics
- OpenMP: The more subtle/advanced stuff
- ➡ OpenMP case studies
  - Parallelization of the SPEC OMP 2001 benchmarks
  - Performance tuning method
- Automatic parallelism and tools support
- Mixing OpenMP and MPI
- The future of OpenMP

The SPEC OMP2001 Applications

Code	Applications	Language	lines
ammp	Chemistry/biology	C	13500
applu	Fluid dynamics/physics	Fortran	4000
apsi	Air pollution	Fortran	7500
art	Image Recognition/ neural networks	C	1300
fma3d	Crash simulation	Fortran	60000
gafort	Genetic algorithm	Fortran	1500
galgel	Fluid dynamics	Fortran	15300
equake	Earthquake modeling	C	1500
mgrid	Multigrid solver	Fortran	500
swim	Shallow water modeling	Fortran	400
wupwise	Quantum chromodynamics	Fortran	2200

Basic Characteristics

Code	Parallel Coverage (%)	Total Runtime (sec)		# of parallel regions
		Seq.	4-cpu	
ammp	99.11	16841	5898	7
applu	99.99	11712	3677	22
apsi	99.84	8969	3311	24
art	99.82	28008	7698	3
equake	99.15	6953	2806	11
fma3d	99.45	14852	6050	92/30*
gafort	99.94	19651	7613	6
galgel	95.57	4720	3992	31/32*
mgrid	99.98	22725	8050	12
swim	99.44	12920	7613	8
wupwise	99.83	19250	5788	10

* lexical parallel regions / parallel regions called at runtime

Wupwise

- Quantum chromodynamics model written in Fortran 90
- Parallelization was relatively straightforward
  - 10 OMP PARALLEL regions
  - PRIVATE and (2) REDUCTION clauses
  - 1 critical section
- Loop coalescing was used to increase the size of parallel sections

Major parallel loop in Wupwise

Logic added to support loop collalescing

```
C$OMP PARALLEL
C$OMP+      PRIVATE (AUX1, AUX2, AUX3),
C$OMP+      PRIVATE (I, IM, IP, J, JM, JP, K, KM, KP, L, LM, LP),
C$OMP+      SHARED (N1, N2, N3, N4, RESULT, U, X)

C$OMP DO
   DO 100 JKL = 0, N2 * N3 * N4 - 1

     L = MOD (JKL / (N2 * N3), N4) + 1
     LP=MOD(L,N4)+1

     K = MOD (JKL / N2, N3) + 1
     KP=MOD(K,N3)+1

     J = MOD (JKL, N2) + 1
     JP=MOD(J,N2)+1

     DO 100 I=(MOD(J+K+L,2)+1),N1,2

        IP=MOD(I,N1)+1

       CALL GAMMUL(1,0,X(1,(IP+1)/2,J,K,L),AUX1)
       CALL SU3MUL(U(1,1,1,I,J,K,L),'N',AUX1,AUX3)

       CALL GAMMUL(2,0,X(1,(I+1)/2,JP,K,L),AUX1)
       CALL SU3MUL(U(1,1,2,I,J,K,L),'N',AUX1,AUX2)
       CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)

       CALL GAMMUL(3,0,X(1,(I+1)/2,J,KP,L),AUX1)
       CALL SU3MUL(U(1,1,3,I,J,K,L),'N',AUX1,AUX2)
       CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)

       CALL GAMMUL(4,0,X(1,(I+1)/2,J,K,LP),AUX1)
       CALL SU3MUL(U(1,1,4,I,J,K,L),'N',AUX1,AUX2)
       CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)

       CALL ZCOPY(12,AUX3,1,RESULT(1,(I+1)/2,J,K,L),1)

 100  CONTINUE
C$OMP END DO
C$OMP END PARALLEL
```

Swim

- Shallow Water model written in F77/F90
- Swim is known to be highly parallel
- Code contains several doubly-nested loops
  The outer loops are parallelized

Example parallel loop

```
!$OMP PARALLEL DO
      DO 100 J=1,N
      DO 100 I=1,M
      CU(I+1,J) = .5D0*(P(I+1,J)+P(I,J))*U(I+1,J)
      CV(I,J+1) = .5D0*(P(I,J+1)+P(I,J))*V(I,J+1)
      Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
                         -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
      H(I,J) = P(I,J)+.25D0*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
                         +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
  100 CONTINUE
```

Mgrid

- Multigrid electromagnetism in F77/F90
- Major parallel regions inrprj3, basic multigrid iteration
- Simple loop nest patterns, similar to Swim, several 3-nested loops
- Parallelized through the Polaris automatic parallelizing source-to-source translator

Applu

- Non-linear PDES time stepping SSOR in F77
- Major parallel regions in ssor.f, basic SSOR iteration
- Basic parallelization over the outer of 3D loop, temporaries held private

Up to 4-nested loops:

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(M,I,J,K,tmp2)
        tmp2 = dt
!$omp do
        do k = 2, nz - 1
           do j = jst, jend
              do i = ist, iend
                 do m = 1, 5
                    rsd(m,i,j,k) = tmp2 * rsd(m,i,j,k)
                 end do
              end do
           end do
        end do
!$omp end do
!$OMP END PARALLEL
```

Galgel

- CFD in F77/F90
- Major parallel regions in heat transfer calculation
- Loop coalescing applied to increase parallel regions, guided self scheduling in loop with irregular iteration times

Major parallel loop in subroutine syshtN.f of Galgel

```
!$OMP PARALLEL
!$OMP+  DEFAULT(NONE)
!$OMP+  PRIVATE (I, IL, J, JL, L, LM, M, LPOP, LPOP1),
!$OMP+  SHARED (DX, HtTim, K, N, NKX, NKY, NX, NY, Poj3, Poj4, XP, Y),
!$OMP+  SHARED (WXXX, WXXY, WXYX, WXYY, WYXX, WYXY, WYYX, WYYY),
!$OMP+  SHARED (WXTX, WYTX, WXTY, WYTY, A, Ind0)
          If (Ind0 .NE. 1) then
                              ! Calculate r.h.s.

C ++++++ - HtCon(i,j,l)*Z(j)*X(l) +++++++++++++++++++++++++++++++++++++

!$OMP DO SCHEDULE(GUIDED)
               Ext12: Do LM = 1, K
                L = (LM - 1) / NKY + 1
                M = LM - (L - 1) * NKY

                Do IL=1,NX
                 Do JL=1,NY
                  Do i=1,NKX
                   Do j=1,NKY

                     LPOP( NKY*(i-1)+j, NY*(IL-1)+JL ) =
                              WXTX(IL,i,L) * WXTY(JL,j,M) + WYTX(IL,i,L) * WYTY(JL,j,M)
                   End Do
                  End Do
                 End Do
                End Do

C ............. LPOP1(i) = LPOP(i,j)*X(j) ..........................

                LPOP1(1:K) = MATMUL( LPOP(1:K,1:N), Y(K+1:K+N) )

C ............. Poj3 = LPOP1 .....................................

                Poj3( NKY*(L-1)+M, 1:K) = LPOP1(1:K)

C .............. Xp = <LPOP1,Z> ..................................

               Xp(NKY*(L-1)+M) =  DOT_PRODUCT (Y(1:K), LPOP1(1:K) )

C .............. Poj4(*,i) = LPOP(j,i)*Z(j) ........................

                Poj4( NKY*(L-1)+M,1:N) =
                             MATMUL( TRANSPOSE( LPOP(1:K,1:N) ), Y(1:K) )

               End Do Ext12
!$OMP END DO
```

```
C ........... DX = DX - HtTim*Xp ..........................
!$OMP DO
                DO LM = 1, K
                 DX(LM) = DX(LM) - DOT_PRODUCT (HtTim(LM,1:K), Xp(1:K))
                END DO
!$OMP END DO NOWAIT

          Else

C ************ Jacobian ****************************************

C ...........A = A - HtTim * Poj3 ......................

!$OMP DO
                DO LM = 1, K
                A(1:K,LM) = A(1:K,LM) -
                                   MATMUL( HtTim(1:K,1:K), Poj3(1:K,LM) )
                END DO
!$OMP END DO NOWAIT

C ...........A = A - HtTim * Poj4 ......................

!$OMP DO
                DO LM = 1, N
                A(1:K,K+LM) = A(1:K,K+LM) -
                                  MATMUL( HtTim(1:K,1:K), Poj4(1:K,LM) )
                END DO
!$OMP END DO NOWAIT

          End If
!$OMP END PARALLEL

      Return
      End
```

APSI

- 3D air pollution model
- Relatively flat profile
- Parts of work arrays used as shared and other parts used as private data

Sample parallel loop from run.f

```
!$OMP PARALLEL
!$OMP+PRIVATE(II,MLAG,HELP1,HELPA1)
!$OMP DO
      DO 20 II=1,NZTOP
        MLAG=NXNY1+II*NXNY
C
C              HORIZONTAL DISPERSION PART          2    2    2    2
C ----   CALCULATE WITH  DIFFUSION EIGENVALUES THE  K D C/DX ,K D C/DY
C                                             X         Y
        CALL DCTDX(NX,NY,NX1,NFILT,C(MLAG),DCDX(MLAG),
                     HELP1,HELPA1,FX,FXC,SAVEX)
        IF(NY.GT.1) CALL DCTDY(NX,NY,NY1,NFILT,C(MLAG),DCDY(MLAG),
                                    HELP1,HELPA1,FY,FYC,SAVEY)
 20  CONTINUE
!$OMP END DO
!$OMP END PARALLEL
```

Gafort

- Genetic algorithm in Fortran
- Most “interesting” loop: shuffle the population.
  - Original loop is not parallel; performs pair-wise swap of an array element with another, randomly selected element. There are 40,000 elements.
  - Parallelization idea:
    - Perform the swaps in parallel
    - Need to prevent simultaneous access to same array element: use one lock per array element → 40,000 locks.

Parallel loop In shuffle.f of Gafort

```
!$OMP PARALLEL PRIVATE(rand, iother, itemp, temp, my_cpu_id)
    my_cpu_id = 1
!$  my_cpu_id = omp_get_thread_num() + 1
!$OMP DO
    DO j=1,npopsiz-1
      CALL ran3(1,rand,my_cpu_id,0)
      iother=j+1+DINT(DBLE(npopsiz-j)*rand)
!$    IF (j < iother) THEN
!$        CALL omp_set_lock(lck(j))
!$        CALL omp_set_lock(lck(iother))
!$    ELSE
!$        CALL omp_set_lock(lck(iother))
!$        CALL omp_set_lock(lck(j))
!$    END IF
      itemp(1:nchrome)=iparent(1:nchrome,iother)
      iparent(1:nchrome,iother)=iparent(1:nchrome,j)
      iparent(1:nchrome,j)=itemp(1:nchrome)
      temp=fitness(iother)
      fitness(iother)=fitness(j)
      fitness(j)=temp
!$    IF (j < iother) THEN
!$        CALL omp_unset_lock(lck(iother))
!$        CALL omp_unset_lock(lck(j))
!$    ELSE
!$        CALL omp_unset_lock(lck(j))
!$        CALL omp_unset_lock(lck(iother))
!$    END IF
   END DO
!$OMP END DO
!$OMP END PARALLEL
```

Exclusive access to array elements. Ordered locking prevents deadlock.

Fma3D

- 3D finite element mechanical simulator
- Largest of the SPEC OMP codes: 60,000 lines
- Uses OMP DO, REDUCTION, NOWAIT, CRITICAL
- Key to good scaling was critical section
- Most parallelism from simple DOs
  - Of the 100 subroutines only four have parallel sections; most of them in fma1.f90
- Conversion to OpenMP took substantial work

Parallel loop in platq.f90 of Fma3D

```
!$OMP PARALLEL DO &
!$OMP   DEFAULT(PRIVATE), SHARED(PLATQ,MOTION,MATERIAL,STATE_VARIABLES), &
!$OMP   SHARED(CONTROL,TIMSIM,NODE,SECTION_2D,TABULATED_FUNCTION,STRESS),&
!$OMP   SHARED(NUMP4) REDUCTION(+:ERRORCOUNT),                  &
!$OMP   REDUCTION(MIN:TIME_STEP_MIN),                  &
!$OMP   REDUCTION(MAX:TIME_STEP_MAX)

    DO N = 1,NUMP4

     ... (66 lines deleted)

      MatID = PLATQ(N)%PAR%MatID

      CALL PLATQ_MASS ( NEL,SecID,MatID )

     ... (35 lines deleted)

      CALL PLATQ_STRESS_INTEGRATION ( NEL,SecID,MatID )

     ... (34 lines deleted)

!$OMP END PARALLEL DO
```

Contains large critical section

Subroutine platq_mass.f90 of Fma3D

```
 SUBROUTINE PLATQ_MASS ( NEL,SecID,MatID )

   ... (54 lines deleted)

!$OMP CRITICAL (PLATQ_MASS_VALUES)
    DO i = 1,4
      NODE(PLATQ(NEL)%PAR%IX(i))%Mass = NODE(PLATQ(NEL)%PAR%IX(i))%Mass + QMass
      MATERIAL(MatID)%Mass = MATERIAL(MatID)%Mass + QMass
      MATERIAL(MatID)%Xcm  = MATERIAL(MatID)%Xcm  + QMass * Px(I)
      MATERIAL(MatID)%Ycm  = MATERIAL(MatID)%Ycm  + QMass * Py(I)
      MATERIAL(MatID)%Zcm  = MATERIAL(MatID)%Zcm  + QMass * Pz(I)
!!
!! Compute inertia tensor B wrt the origin from nodal point masses.
!!
      MATERIAL(MatID)%Bxx = MATERIAL(MatID)%Bxx + (Py(I)*Py(I)+Pz(I)*Pz(I))*QMass
      MATERIAL(MatID)%Byy = MATERIAL(MatID)%Byy + (Px(I)*Px(I)+Pz(I)*Pz(I))*QMass
      MATERIAL(MatID)%Bzz = MATERIAL(MatID)%Bzz + (Px(I)*Px(I)+Py(I)*Py(I))*QMass
      MATERIAL(MatID)%Bxy = MATERIAL(MatID)%Bxy - Px(I)*Py(I)*QMass
      MATERIAL(MatID)%Bxz = MATERIAL(MatID)%Bxz - Px(I)*Pz(I)*QMass
      MATERIAL(MatID)%Byz = MATERIAL(MatID)%Byz - Py(I)*Pz(I)*QMass
    ENDDO
!!
!!
!! Compute nodal isotropic inertia
!!
    RMass = QMass * (PLATQ(NEL)%PAR%Area + SECTION_2D(SecID)%Thickness**2) / 12.0D+0
!!
!!
    NODE(PLATQ(NEL)%PAR%IX(5))%Mass = NODE(PLATQ(NEL)%PAR%IX(5))%Mass + RMass
    NODE(PLATQ(NEL)%PAR%IX(6))%Mass = NODE(PLATQ(NEL)%PAR%IX(6))%Mass + RMass
    NODE(PLATQ(NEL)%PAR%IX(7))%Mass = NODE(PLATQ(NEL)%PAR%IX(7))%Mass + RMass
    NODE(PLATQ(NEL)%PAR%IX(8))%Mass = NODE(PLATQ(NEL)%PAR%IX(8))%Mass + RMass
!$OMP END CRITICAL (PLATQ_MASS_VALUES)
!!
!!
    RETURN
    END
```

This is a large array reduction

Art

- Image processing
- Good scaling required combining two dimensions into single dimension
- Uses OMP DO, SCHEDULE(DYNAMIC)
- Dynamic schedule needed because of embedded conditional

Loop collalescing

Key loop in Art

```
#pragma omp for private (k,m,n, gPassFlag) schedule(dynamic)
   for (ij = 0; ij < ijmx; ij++)  {
      j = ((ij/inum) * gStride) + gStartY;
      i = ((ij%inum) * gStride) +gStartX;
      k=0;
      for (m=j;m<(gLheight+j);m++)
        for (n=i;n<(gLwidth+i);n++)
          f1_layer[o][k++].I[0] = cimage[m][n];

      gPassFlag =0;
      gPassFlag = match(o,i,j, &mat_con[ij], busp);

      if (gPassFlag==1) {
         if (set_high[o][0]==TRUE) {
           highx[o][0] = i;
           highy[o][0] = j;
           set_high[o][0] = FALSE;
         }
       if (set_high[o][1]==TRUE)  {
         highx[o][1] = i;
         highy[o][1] = j;
         set_high[o][1] = FALSE;
       }
      }
   }
```

Ammp

- Molecular Dynamics
- Very large loop in rectmm.c
- Good parallelism required great deal of work
- Uses OMP FOR, SCHEDULE(GUIDED), about 20,000 locks
- Guided scheduling needed because of loop with conditional execution.

Parallel loop in rectmm.c of Ammp

```
#pragma omp parallel for private (n27ng0, nng0, ing0, i27ng0, natoms, ii, a1, a1q, a1serial,
   inclose, ix, iy, iz, inode, nodelistt, r0, r, xt, yt, zt, xt2, yt2, zt2, xt3, yt3, zt3, xt4,
   yt4, zt4, c1, c2, c3, c4, c5, k, a1VP , a1dpx , a1dpy , a1dpz , a1px, a1py, a1pz, a1qxx ,
   a1qxy , a1qxz ,a1qyy , a1qyz , a1qzz, a1a, a1b, iii, i, a2, j, k1, k2 ,ka2, kb2, v0, v1, v2,
   v3, kk, atomwho, ia27ng0, iang0,  o ) schedule(guided)

    for( ii=0; ii<  jj; ii++)
    ...
                  for( inode = 0; inode < iii; inode ++)
                     if( (*nodelistt)[inode].innode > 0) {
                       for(j=0; j< 27; j++)
                       if( j == 27  )
    ...
                                   if(  atomwho->serial > a1serial)
                                                 for( kk=0; kk< a1->dontuse; kk++)
                                                    if( atomwho == a1->excluded[kk])
    ...
                                   for( j=1; j< (*nodelistt)[inode].innode -1 ; j++)
    ...
                                                 if( atomwho->serial > a1serial)
                                                    for( kk=0; kk< a1->dontuse; kk++)
                                                                  if( atomwho == a1->excluded[kk]) goto SKIP2;
    ...
                  for (i27ng0=0 ; i27ng0<n27ng0; i27ng0++)
    ...
    ...
                  for( i=0; i< nng0; i++)
    ...
                     if( v3 > mxcut || inclose > NCLOSE )
    ...
    ...
```

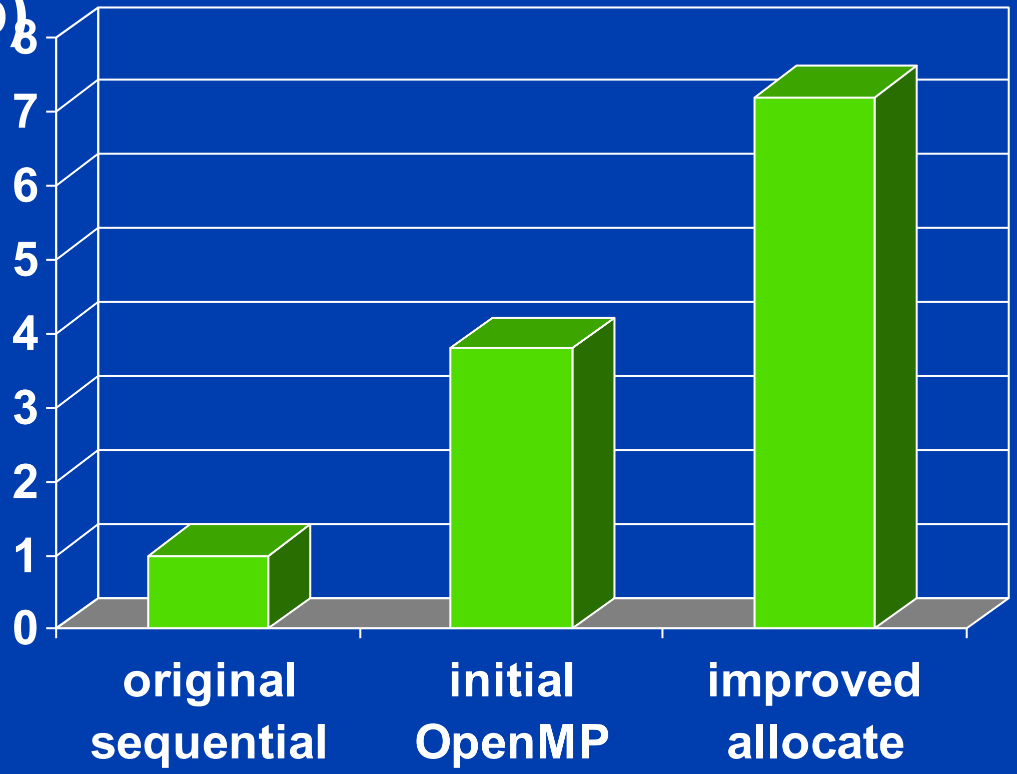
(loop body contains 721 lines)

Performance Tuning Example 3: EQUAKE

EQUAKE: Earthquake simulator in C

(run on a 4 processor SUN Enterprise system – note super linear speedup)

EQUAKE is hand-parallelized with relatively few code modifications.



EQUAKE: Tuning Steps

- Step1:

  Parallelizing the four most time-consuming loops

  - inserted OpenMP pragmas for parallel loops and private data
  - array reduction transformation

- Step2:

  A change in memory allocation

EQUAKE Code Samples

```
 /* malloc w1[numthreads][ARCHnodes][3] */

#pragma omp parallel for
  for (j = 0; j < numthreads; j++)
    for (i = 0; i < nodes; i++) { w1[j][i][0] = 0.0; ...; }

#pragma omp parallel private(my_cpu_id,exp,...)
{
  my_cpu_id = omp_get_thread_num();

#pragma omp for
  for (i = 0; i < nodes; i++)
    while (...) {
      ...
      exp = loop-local computation;
      w1[my_cpu_id][...][1] += exp;
      ...
  }
}
#pragma omp parallel for
  for (j = 0; j < numthreads; j++) {
    for (i = 0; i < nodes; i++) { w[i][0] += w1[j][i]
[0]; ...;}
```

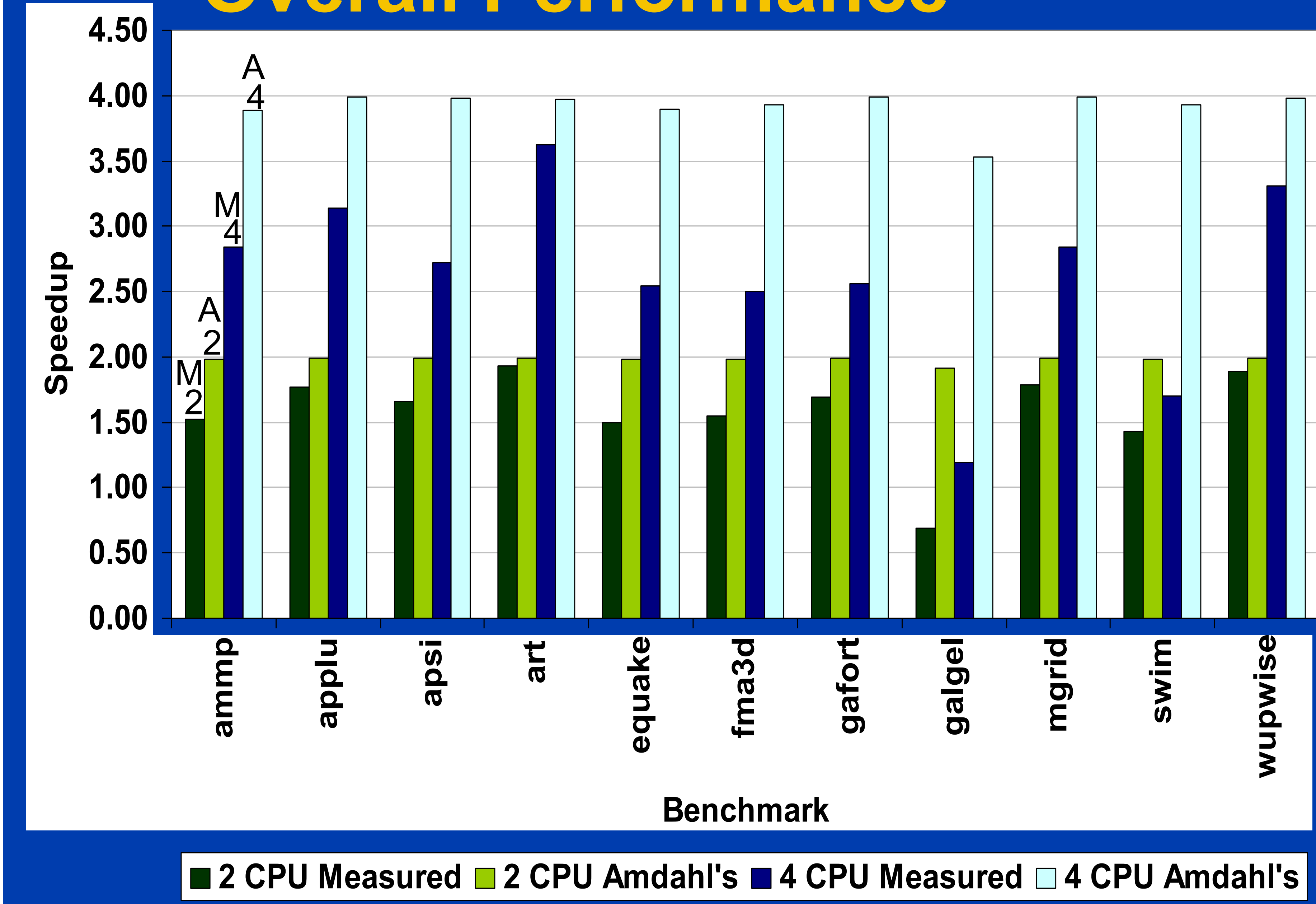
OpenMP Features Used

Code	sections	locks	guided	dynamic	critical	nowait
ammp	7	20k	2			
applu	22					14
apsi	24					
art	3			1		
equake	11					
fma3d	92/30				1	2
gafort	6	40k				
galgel	31/32*		7			3
mgrid	12					11
swim	8					
wupwise	10				1	

* static sections / sections called at runtime

"Feature" used to deal with NUMA machines: rely on *first-touch* page placement. If necessary, put initialization into a parallel loop to avoid placing all data on the master processor.

Overall Performance



What Tools Did We Use for Performance Analysis and Tuning?

- Compilers
  - for several applications, the starting point for our performance tuning of Fortran codes was the compiler-parallelized program.
  - It reports: parallelized loops, data dependences.
- Subroutine and loop profilers
  - focusing attention on the most time-consuming loops is absolutely essential.
- Performance tables:
  - typically comparing performance differences at the loop level.

Guidelines for Fixing "Performance Bugs"

- The methodology that worked for us:
  - Use compiler-parallelized code as a starting point
  - Get loop profile and compiler listing
  - Inspect time-consuming loops (biggest potential for improvement)
    - Case 1. Check for parallelism where the compiler could not find it
    - Case 2. Improve parallel loops where the speedup is limited

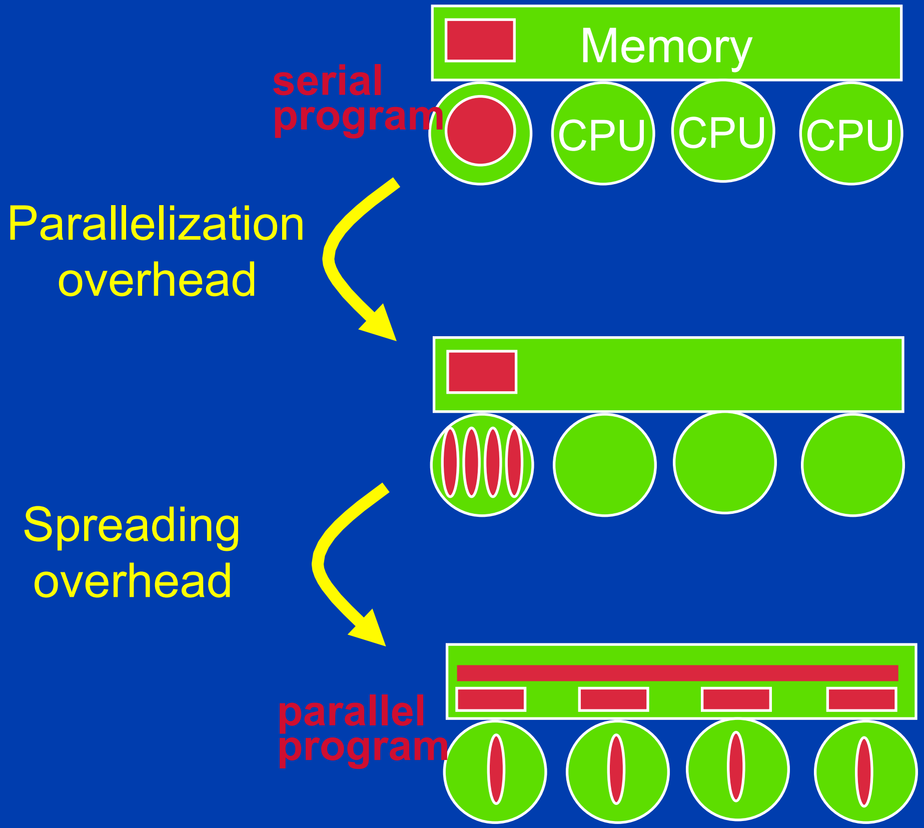
Performance Tuning

Case 1: if the loop is not yet parallelized, do this:

- Check for parallelism:
  - read the compiler explanation
  - a variable may be independent even if the compiler detects dependences (compilers are conservative)
  - check if conflicting array is privatizable (compilers don’t perform array privatization well)
- If you find parallelism, add OpenMP parallel directives, or make the information explicit for the parallelizer

Performance Tuning

Case 2: if the loop is parallel but does not perform well, consider several optimization factors:



High overheads are caused by:

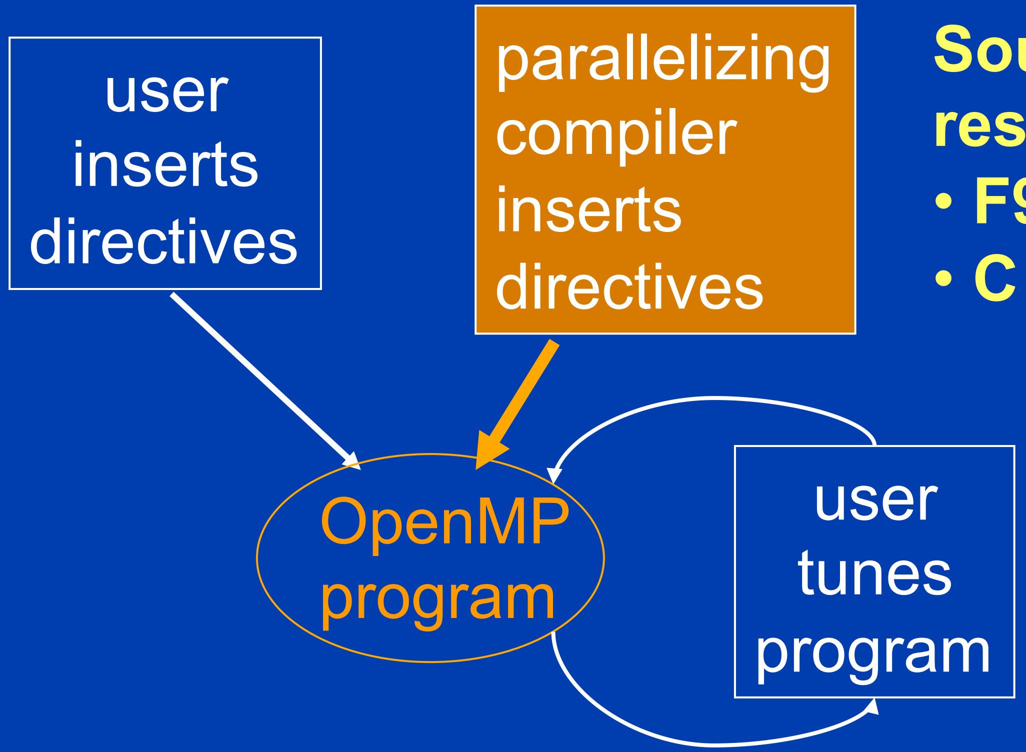
- parallel startup cost
- small loops
- additional parallel code
- over-optimized inner loops
- less optimization for parallel code

- load imbalance
- synchronized section
- non-stride-1 references
- many shared references
- low cache affinity

Agenda

- Summary of OpenMP basics
- OpenMP: The more subtle/advanced stuff
- OpenMP case studies
- ➡ Automatic parallelism and tools support
- Mixing OpenMP and MPI
- The future of OpenMP

Generating OpenMP Programs Automatically



Source-to-source restructurers:
- **F90 to F90/OpenMP**
- **C to C/OpenMP**

Examples:
- SGI F77 compiler (-apo -mplist option)
- Polaris compiler

The Basics About Parallelizing Compilers

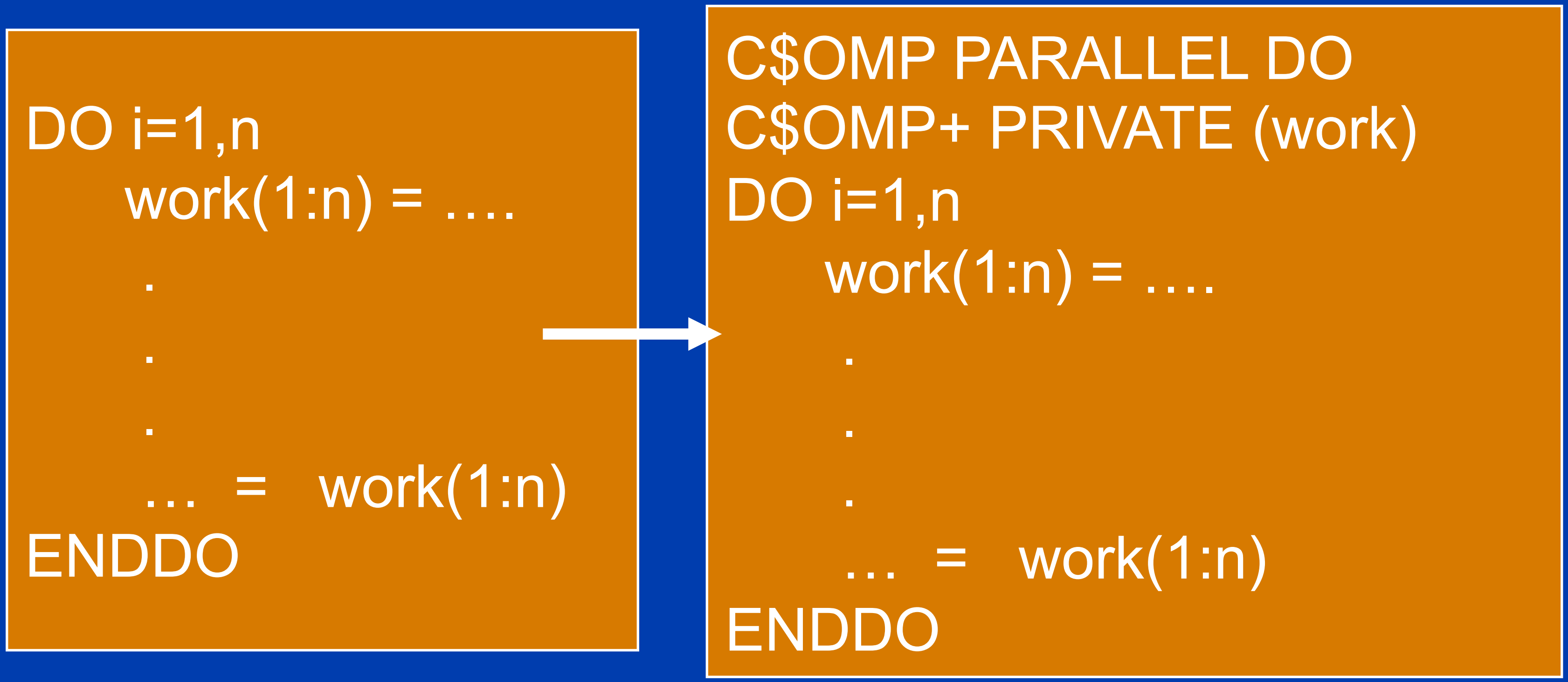
- Loops are the primary source of parallelism in scientific and engineering applications.
- Compilers detect loops that have independent iterations.

```
DO I=1,N
    A(expression1) = …
        … = A(expression2)
ENDDO
```

The loop is independent if, for different iterations, *expression1* is always different from *expression2*

Basic Program Transformations

Data privatization:



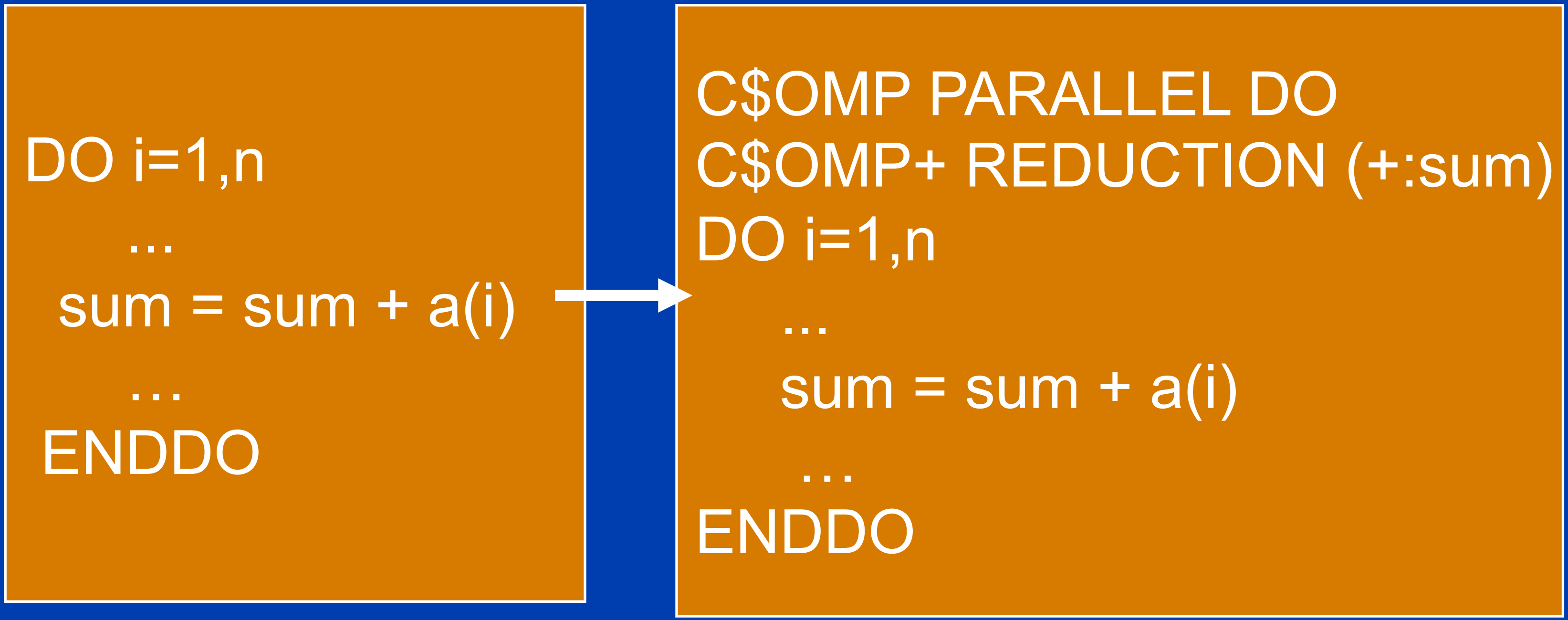
```
DO i=1,n
      work(1:n) = ….
       .
       .
       .
       …  =   work(1:n)
ENDDO
```

```
C$OMP PARALLEL DO
C$OMP+ PRIVATE (work)
DO i=1,n
      work(1:n) = ….
       .
       .
       .
       …  =   work(1:n)
ENDDO
```

Each processor is given a separate version of the private data, so there is no sharing conflict

Basic Program Transformations

Reduction recognition:



```
DO i=1,n
     ...
 sum = sum + a(i)
     …
 ENDDO
```

```
C$OMP PARALLEL DO
C$OMP+ REDUCTION (+:sum)
DO i=1,n
     ...
     sum = sum + a(i)
      …
ENDDO
```

Each processor will accumulate partial sums, followed by a combination of these parts at the end of the loop.

Basic Program Transformations

Induction variable substitution:

```
i1 = 0
i2 = 0
DO i =1,n
    i1 = i1 + 1
    B(i1) = ...

    i2 = i2 + i
    A(i2) = …

 ENDDO
```

```

C$OMP PARALLEL DO
DO i =1,n

    B(i) = ...

    A((i**2 + i)/2) = …

ENDDO
```

The original loop contains data dependences: each processor modifies the shared variables *i1*, and *i2*.

Compiler Options

Examples of options from the KAP parallelizing compiler (KAP includes some 60 options)

- **optimization levels**
  - **optimize :** simple analysis, advanced analysis, loop interchanging, array expansion
  - **aggressive:** pad common blocks, adjust data layout
- **subroutine inline expansion**
  - inline all, specific routines, how to deal with libraries
- **try specific optimizations**
  - e.g., recurrence and reduction recognition, loop fusion

(These transformations may degrade performance**)**

More About Compiler Options

- **Limits on amount of optimization:**
  - e.g., size of optimization data structures, number of optimization variants tried
- **Make certain assumptions:**
  - e.g., array bounds are not violated, arrays are not aliased
- **Machine parameters:**
  - e.g., cache size, line size, mapping
- **Listing control**

Note, compiler options can be a substitute for advanced compiler strategies. If the compiler has limited information, the user can help out.

Inspecting the Translated Program

- Source-to-source restructurers:
  - transformed source code is the actual output
  - Example: KAP
- Code-generating compilers:
  - typically have an option for viewing the translated (parallel) code
  - Example: SGI f77 -apo -mplist

This can be the starting point for code tuning

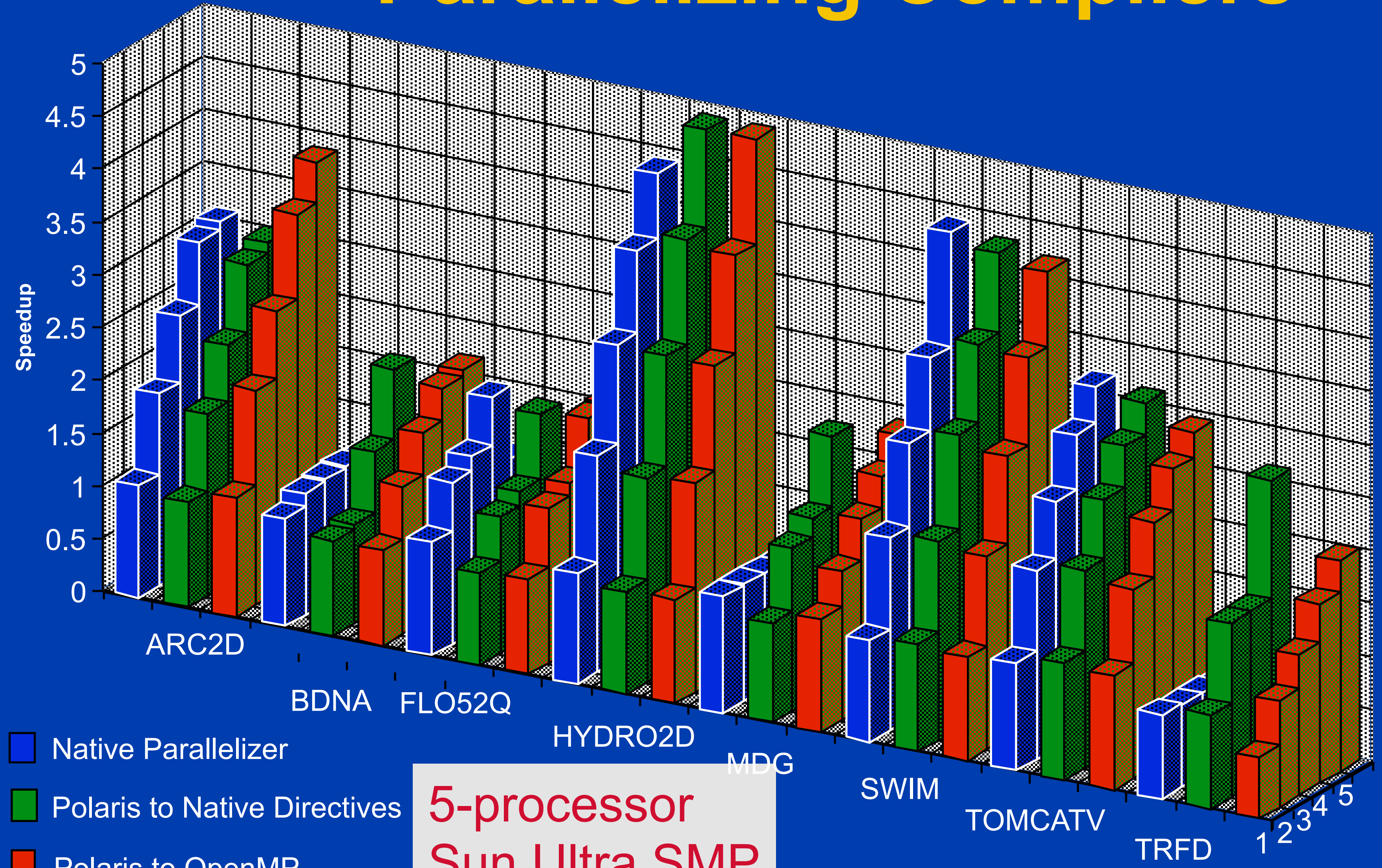
Compiler Listing

The listing gives many useful clues for improving the performance:

- ◆ Loop optimization tables
- ◆ Reports about data dependences
- ◆ Explanations about applied transformations
- ◆ The annotated, transformed code
- ◆ Calling tree
- ◆ Performance statistics

The type of reports to be included in the listing can be set through compiler options.

Performance of Parallelizing Compilers



5-processor
Sun Ultra SMP

Tuning Automatically-Parallelized Code

- This task is similar to explicit parallel programming.
- Two important differences :
  - The compiler gives hints in its listing, which may tell you where to focus attention. E.g., which variables have data dependences.
  - You don’t need to perform all transformations by hand. If you expose the right information to the compiler, it will do the translation for you.

    (E.g., `C$assert independent`)

Why Tuning Automatically-Parallelized Code?

Hand improvements can pay off because

- compiler techniques are limited

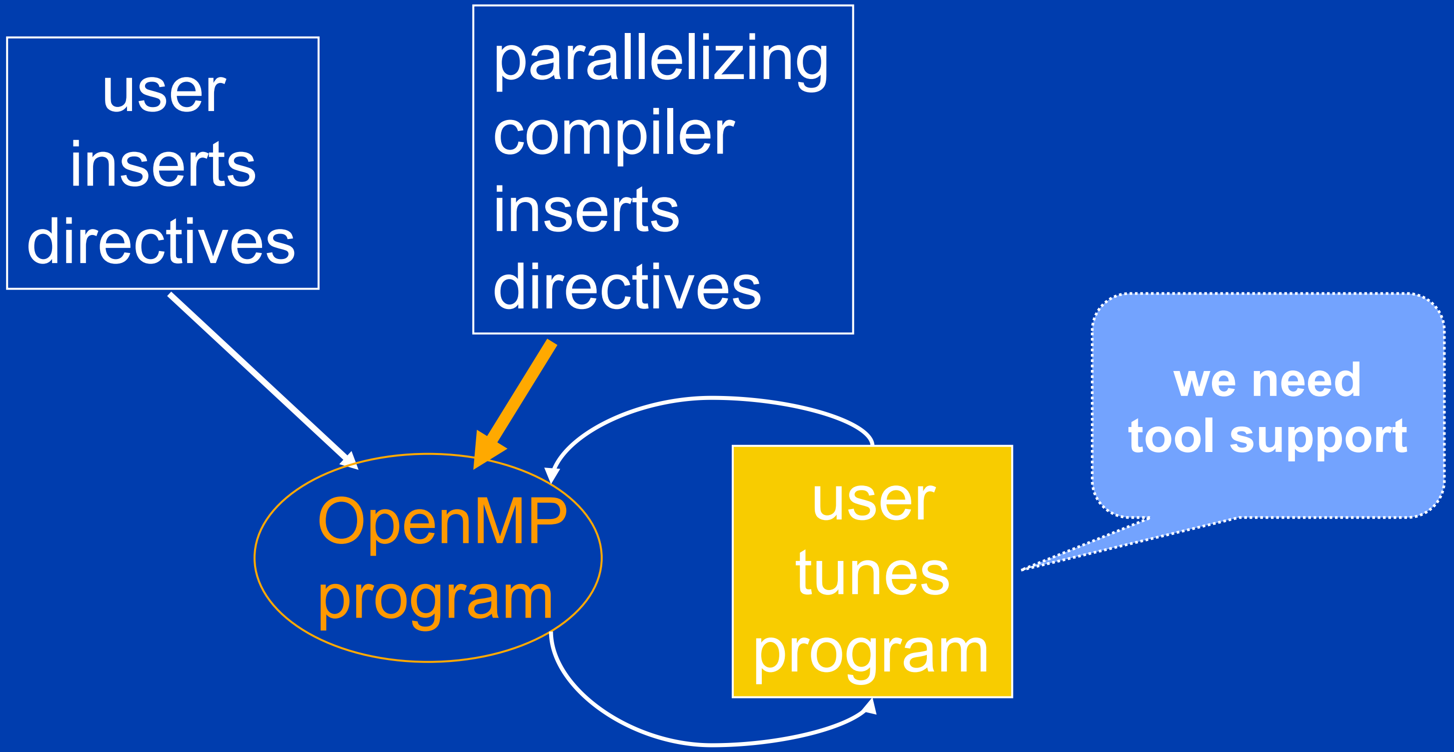
  E.g., array reductions are parallelized by only few compilers

- compilers may have insufficient information

  E.g.,

  - ◆loop iteration range may be input data
  - ◆variables are defined in other subroutines (no interprocedural analysis)

Performance Tuning Tools



Profiling Tools

- Timing profiles (subroutine or loop level)
  - shows most time-consuming program sections
- Cache profiles
  - point out memory/cache performance problems
- Data-reference and transfer volumes
  - show performance-critical program properties
- Input/output activities
  - point out possible I/O bottlenecks
- Hardware counter profiles
  - large number of processor statistics

KAI GuideView: Performance Analysis

- Speedup curves
  - Amdahl's Law vs. Actual times
- Whole program time breakdown
  - Productive work vs
  - Parallel overheads
- Compare several runs
  - Scaling processors
- Breakdown by section
  - Parallel regions
  - Barrier sections
  - Serial sections
- Breakdown by thread
- Breakdown overhead
  - Types of runtime calls
  - Frequency and time

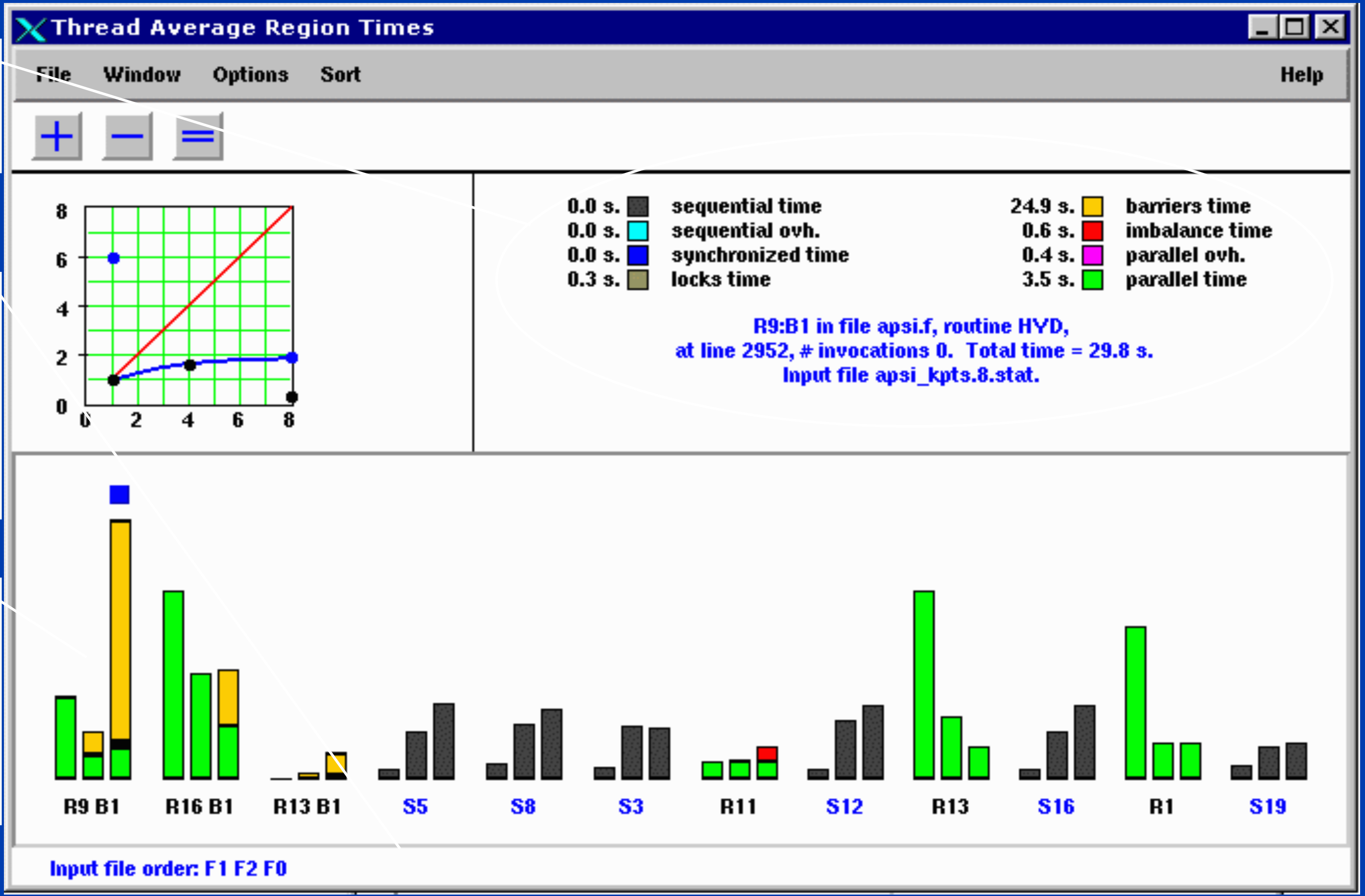
KAI's new VGV tool combines GuideView with VAMPIR for monitoring mixed OpenMP/MPI programs

GuideView

Analyze each Parallel region

Find serial regions that are hurt by parallelism

Sort or filter regions to navigate to hotspots



0.0 s.	■	sequential time	24.9 s.	🟨	barriers time
0.0 s.	■	sequential ovh.	0.6 s.	🟥	imbalance time
0.0 s.	🟦	synchronized time	0.4 s.	■	parallel ovh.
0.3 s.	■	locks time	3.5 s.	🟩	parallel time

R9:B1 in file apsi.f, routine HYD,
at line 2952, # invocations 0. Total time = 29.8 s.
Input file apsi_kpts.8.stat.

Input file order: F1 F2 F0

www.kai.com

SGI SpeedShop and WorkShop

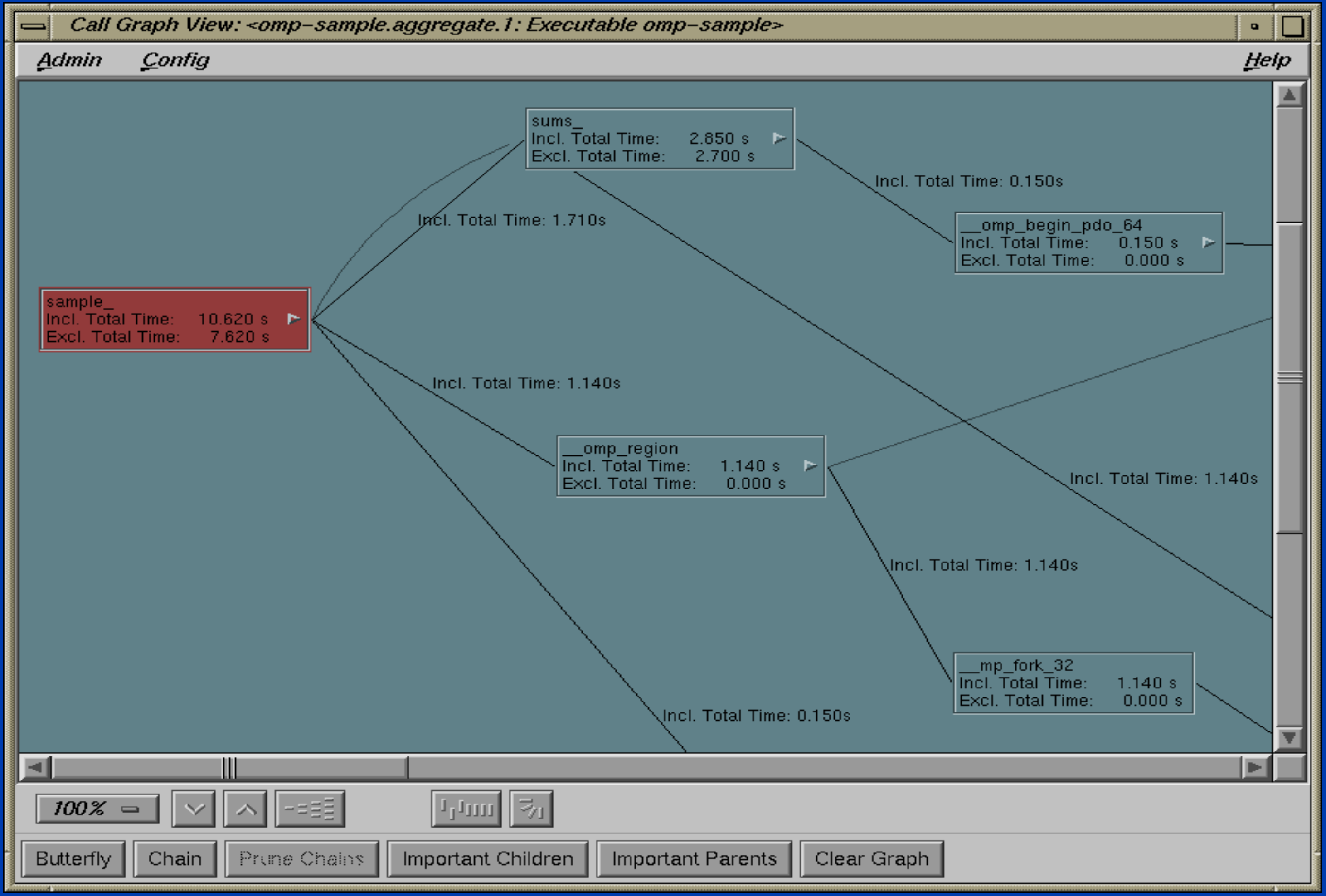
- Suite of performance tools from SGI
- Measurements based on
  - pc-sampling and call-stack sampling
    - based on time [*prof*,*gprof*]
    - based on R10K/R12K hw counters
  - basic block counting [*pixie*]
- Analysis on various domains
  - program graph, source and disassembled code
  - per-thread as well as cumulative data

SpeedShop and WorkShop

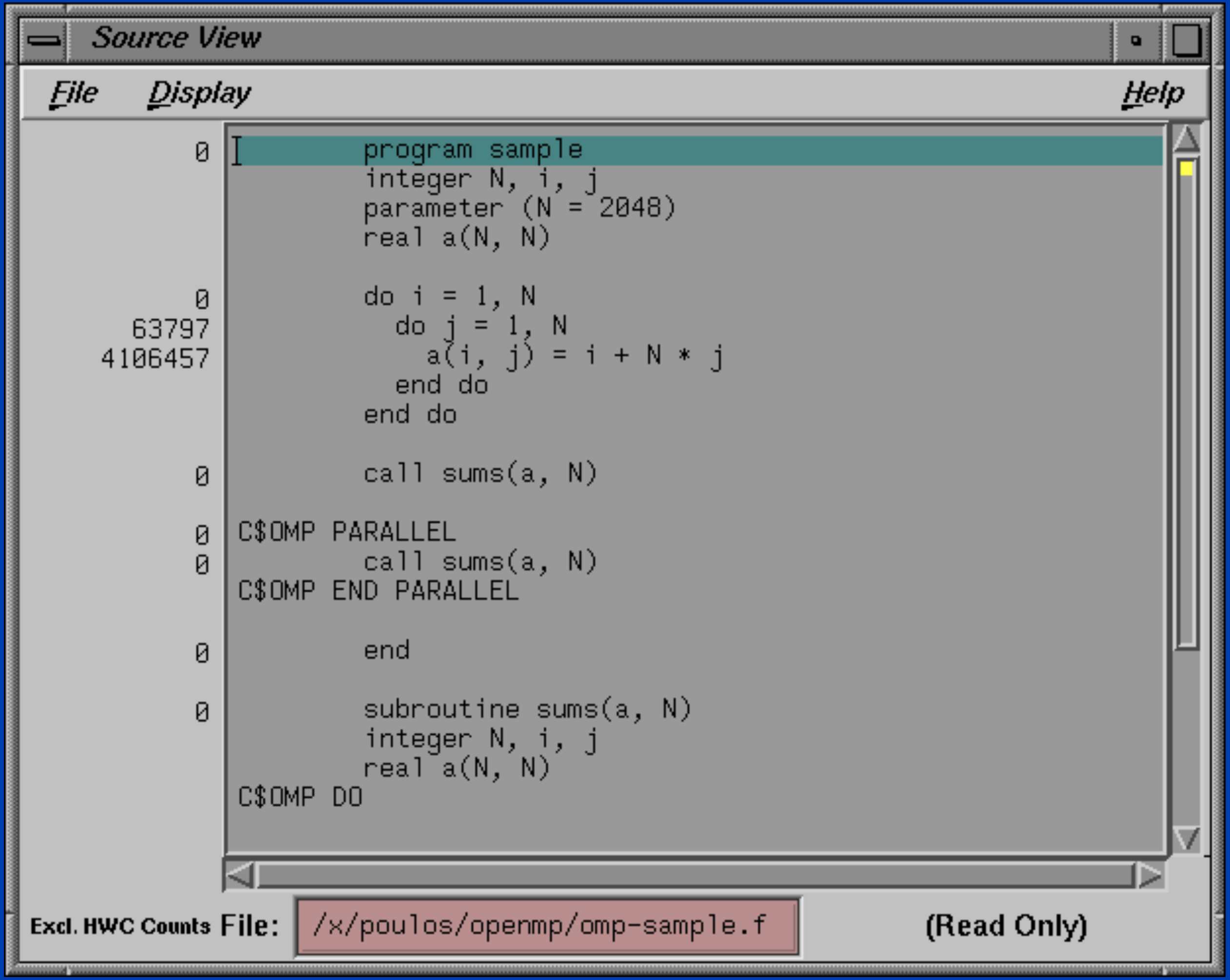
Addresses the performance Issues:

- Load imbalance
  - Call stack sampling based on time (*gprof*)
- Synchronization Overhead
  - Call stack sampling based on time (*gprof*)
  - Call stack sampling based on hardware counters
- Memory Hierarchy Performance
  - Call stack sampling based on hardware counters

WorkShop: Call Graph View


Call Graph View: <omp-sample.aggregate.1: Executable omp-sample>
Admin
Config
Help
sums_
Incl. Total Time: 2.850 s
Excl. Total Time: 2.700 s
Incl. Total Time: 0.150s
__omp_begin_pdo_64
Incl. Total Time: 0.150 s
Excl. Total Time: 0.000 s
Incl. Total Time: 1.710s
sample_
Incl. Total Time: 10.620 s
Excl. Total Time: 7.620 s
Incl. Total Time: 1.140s
__omp_region
Incl. Total Time: 1.140 s
Excl. Total Time: 0.000 s
Incl. Total Time: 1.140s
Incl. Total Time: 1.140s
__mp_fork_32
Incl. Total Time: 1.140 s
Excl. Total Time: 0.000 s
Incl. Total Time: 0.150s
100%
Butterfly
Chain
Prune Chains
Important Children
Important Parents
Clear Graph

WorkShop: Source View



```
       0    program sample
            integer N, i, j
            parameter (N = 2048)
            real a(N, N)

       0    do i = 1, N
   63797      do j = 1, N
 4106457        a(i, j) = i + N * j
              end do
            end do

       0    call sums(a, N)

       0  C$OMP PARALLEL
       0    call sums(a, N)
          C$OMP END PARALLEL

       0    end

       0    subroutine sums(a, N)
            integer N, i, j
            real a(N, N)
          C$OMP DO
```

Excl. HWC Counts File: /x/poulos/openmp/omp-sample.f (Read Only)

Purdue Ursa Minor/Major

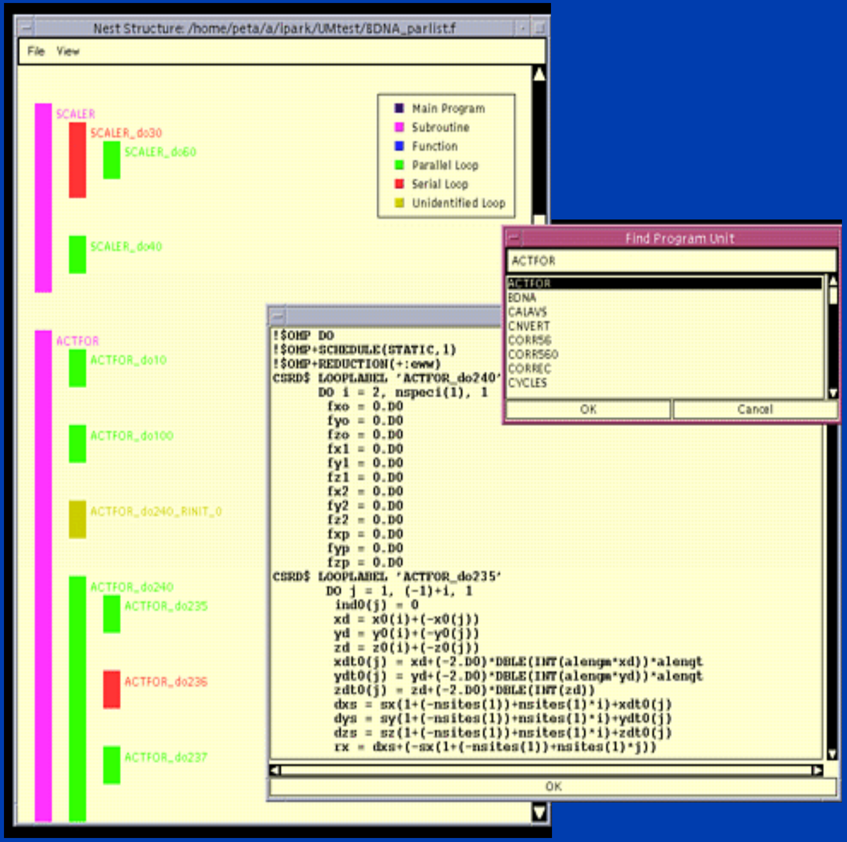
- Integrated environment for compilation and performance analysis/tuning
- Provides browsers for many sources of information:

  call graphs, source and transformed program, compilation reports, timing data, parallelism estimation, data reference patterns, performance advice, etc.

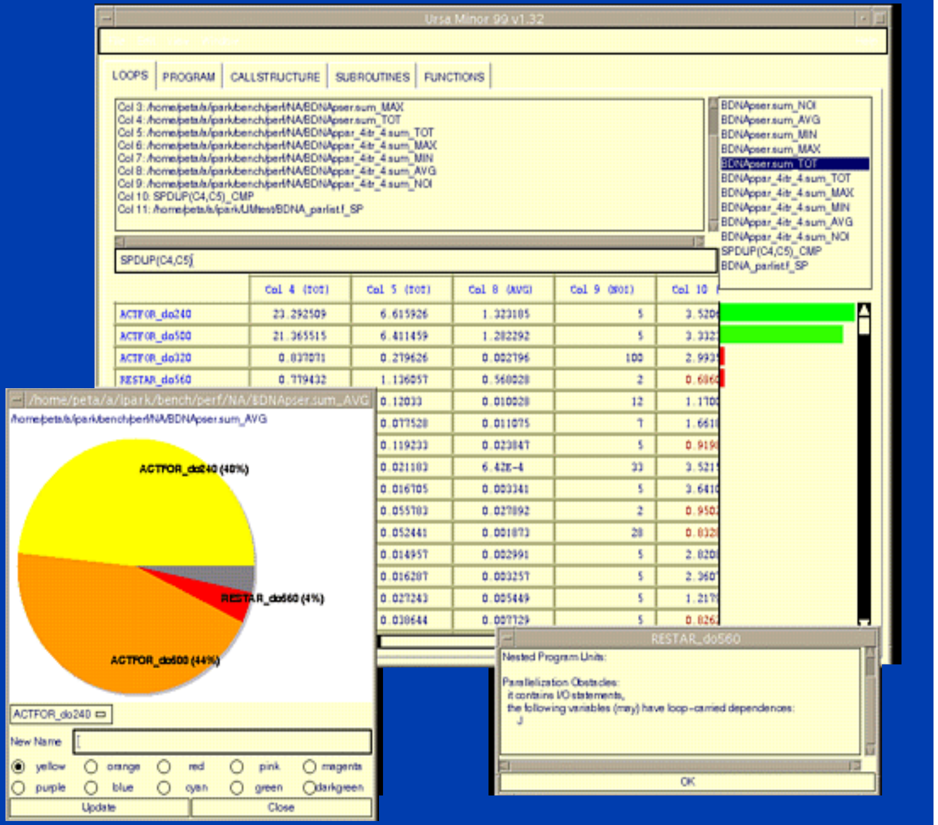
- www.ecn.purdue.edu/ParaMount/UM/

Ursa Minor/Major

Program Structure View



```
!$OMP DO
!$OMP+SCHEDULE(STATIC,1)
!$OMP+REDUCTION(+:eww)
CSRD$ LOOPLABEL 'ACTFOR_do240'
       DO i = 2, nspeci(1), 1
         fxo = 0.D0
         fyo = 0.D0
         fzo = 0.D0
         fx1 = 0.D0
         fy1 = 0.D0
         fz1 = 0.D0
         fx2 = 0.D0
         fy2 = 0.D0
         fz2 = 0.D0
         fxp = 0.D0
         fyp = 0.D0
         fzp = 0.D0
CSRD$ LOOPLABEL 'ACTFOR_do235'
        DO j = 1, (-1)+i, 1
          ind0(j) = 0
          xd = x0(i)+(-x0(j))
          yd = y0(i)+(-y0(j))
          zd = z0(i)+(-z0(j))
          xdt0(j) = xd+(-2.D0)*DBLE(INT(alengm*xd))*alengt
          ydt0(j) = yd+(-2.D0)*DBLE(INT(alengm*yd))*alengt
          zdt0(j) = zd+(-2.D0)*DBLE(INT(zd))
          dxs = sx(1+(-nsites(1))+nsites(1)*i)+xdt0(j)
          dys = sy(1+(-nsites(1))+nsites(1)*i)+ydt0(j)
          dzs = sz(1+(-nsites(1))+nsites(1)*i)+zdt0(j)
          rx = dxs+(-sx(1+(-nsites(1))+nsites(1)*j))
```



Col 3: /home/peta/a/ipark/bench/perf/NA/BDNApser.sum_MAX
Col 4: /home/peta/a/ipark/bench/perf/NA/BDNApser.sum_TOT
Col 5: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_TOT
Col 6: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_MAX
Col 7: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_MIN
Col 8: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_AVG
Col 9: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_NOI
Col 10: SPDUP(C4,C5)_CMP
Col 11: /home/peta/a/ipark/UMtest/BDNA_parlist.f_SP

SPDUP(C4,C5)

	Col 4 (TOT)	Col 5 (TOT)	Col 8 (AVG)	Col 9 (NOI)	Col 10
ACTFOR_do240	23.292509	6.615926	1.323185	5	3.520
ACTFOR_do500	21.365515	6.411459	1.282292	5	3.332
ACTFOR_do320	0.037071	0.279626	0.002796	100	2.993
RESTAR_do560	0.779432	1.136057	0.568028	2	0.686
		0.12033	0.010028	12	1.170
		0.077528	0.011075	7	1.661
		0.119233	0.023847	5	0.919
		0.021103	6.42E-4	33	3.521
		0.016705	0.003341	5	3.641
		0.055703	0.027892	2	0.950
		0.052441	0.001873	28	0.832
		0.014957	0.002991	5	2.820
		0.016287	0.003257	5	2.360
		0.027243	0.005449	5	1.217
		0.038644	0.007729	5	0.826

Nested Program Units:

Parallelization Obstacles:
it contains I/O statements,
the following variables (may) have loop-carried dependences:
J

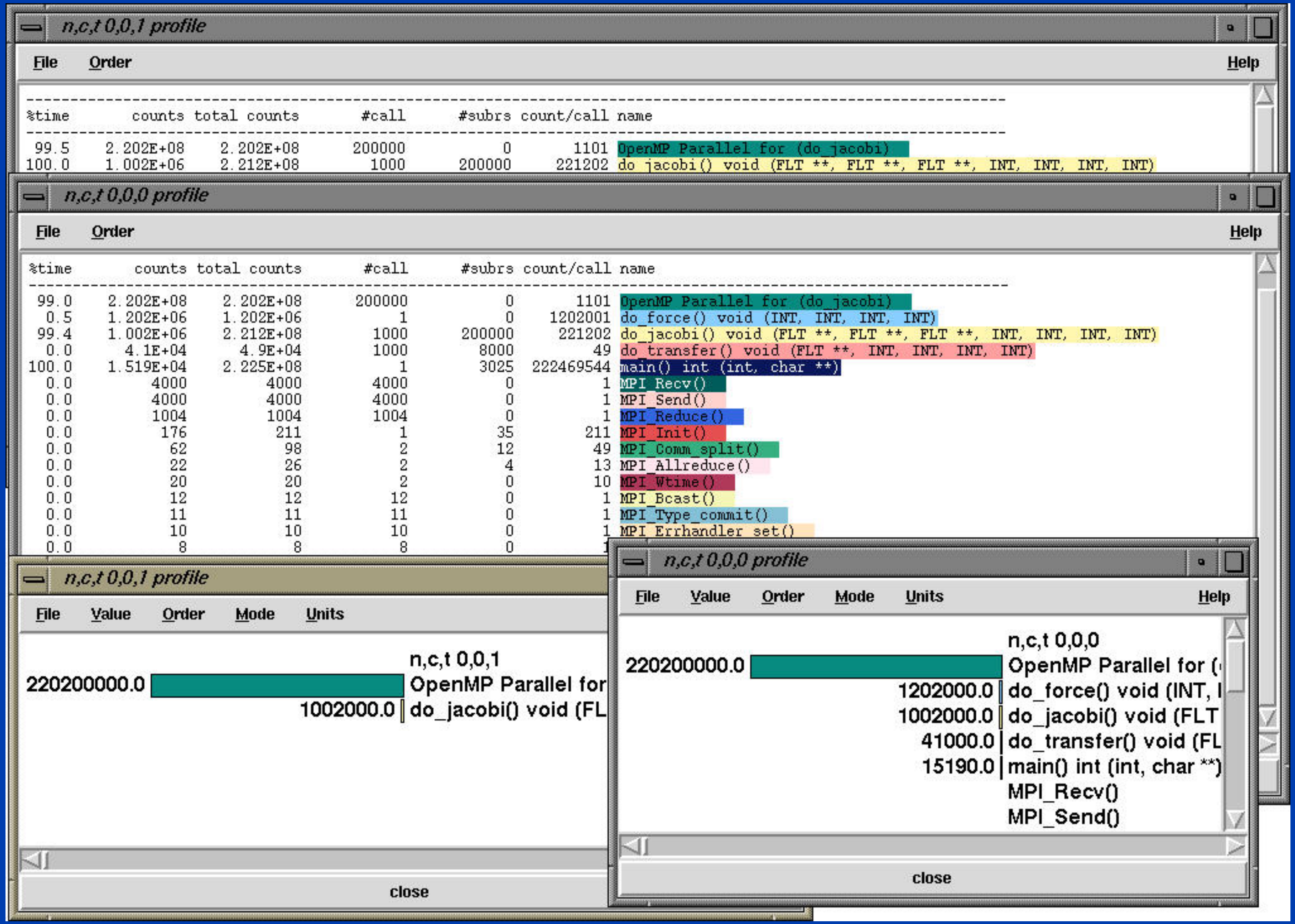
Performance Spreadsheet

TAU
Tuning Analysis Utilities

Performance Analysis Environment for C++, Java, C, Fortran 90, HPF, and HPC++

- compilation facilitator
- call graph browser
- source code browser
- profile browsers
- speedup extrapolation
- www.cs.uoregon.edu/research/paracomp/tau/

TAU
Tuning Analysis Utilities



```
------------------------------------------------------------------------------------------
%time       counts total counts       #call      #subrs count/call name
------------------------------------------------------------------------------------------
 99.5    2.202E+08    2.202E+08      200000           0       1101 OpenMP Parallel for (do_jacobi)
100.0    1.002E+06    2.212E+08        1000      200000     221202 do_jacobi() void (FLT **, FLT **, FLT **, INT, INT, INT, INT)
```

```
%time       counts total counts       #call      #subrs count/call name
------------------------------------------------------------------------------------------
 99.0    2.202E+08    2.202E+08      200000           0       1101 OpenMP Parallel for (do_jacobi)
  0.5    1.202E+06    1.202E+06           1           0    1202001 do_force() void (INT, INT, INT, INT)
 99.4    1.002E+06    2.212E+08        1000      200000     221202 do_jacobi() void (FLT **, FLT **, FLT **, INT, INT, INT, INT)
  0.0      4.1E+04      4.9E+04        1000        8000         49 do_transfer() void (FLT **, INT, INT, INT, INT)
100.0    1.519E+04    2.225E+08           1        3025  222469544 main() int (int, char **)
  0.0         4000         4000        4000           0          1 MPI_Recv()
  0.0         4000         4000        4000           0          1 MPI_Send()
  0.0         1004         1004        1004           0          1 MPI_Reduce()
  0.0          176          211           1          35        211 MPI_Init()
  0.0           62           98           2          12         49 MPI_Comm_split()
  0.0           22           26           2           4         13 MPI_Allreduce()
  0.0           20           20           2           0         10 MPI_Wtime()
  0.0           12           12          12           0          1 MPI_Bcast()
  0.0           11           11          11           0          1 MPI_Type_commit()
  0.0           10           10          10           0          1 MPI_Errhandler_set()
  0.0            8            8           8           0          1
```

```
                              n,c,t 0,0,1
220200000.0                   OpenMP Parallel for
              1002000.0       do_jacobi() void (FL
```

```
                              n,c,t 0,0,0
220200000.0                   OpenMP Parallel for (
              1202000.0       do_force() void (INT, I
              1002000.0       do_jacobi() void (FLT
                41000.0       do_transfer() void (FL
                15190.0       main() int (int, char **)
                              MPI_Recv()
                              MPI_Send()
```

Agenda

- Summary of OpenMP basics
- OpenMP: The more subtle/advanced stuff
- OpenMP case studies
- Automatic parallelism and tools support
- ➡ Mixing OpenMP and MPI
- The future of OpenMP

What is MPI?

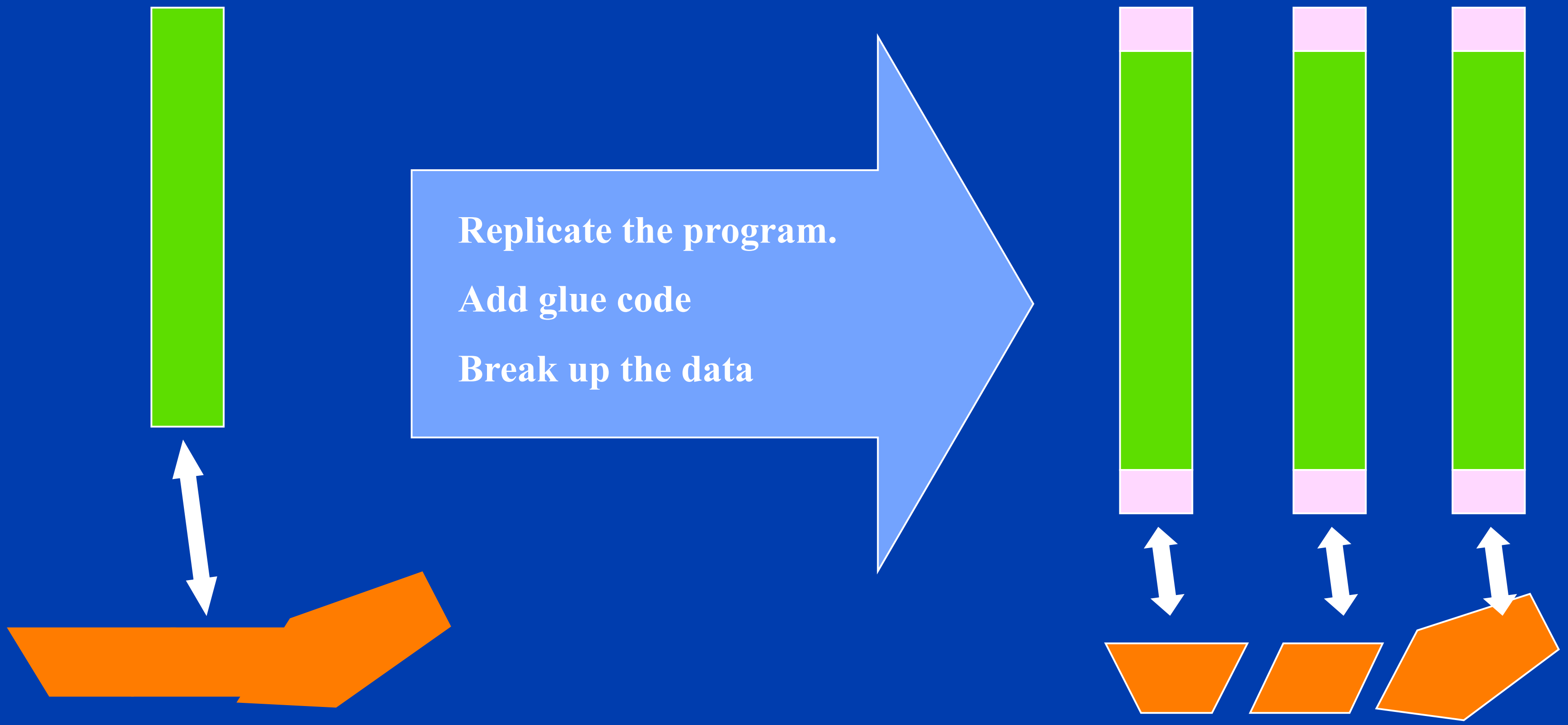
The message Passing Interface

- MPI created by an international forum in the early 90’s.
- It is huge -- the union of many good ideas about message passing API’s.
  - over 500 pages in the spec
  - over 125 routines in MPI 1.1 alone.
  - Possible to write programs using only a couple of dozen of the routines
- MPI 1.1 - MPIch reference implementation.
- MPI 2.0 - Exists as a spec, full implementations? Only one that I know of.

How do people use MPI? The SPMD Model

A sequential program working on a data set

- A parallel program working on a decomposed data set.
- Coordination by passing messages.



Replicate the program.

Add glue code

Break up the data

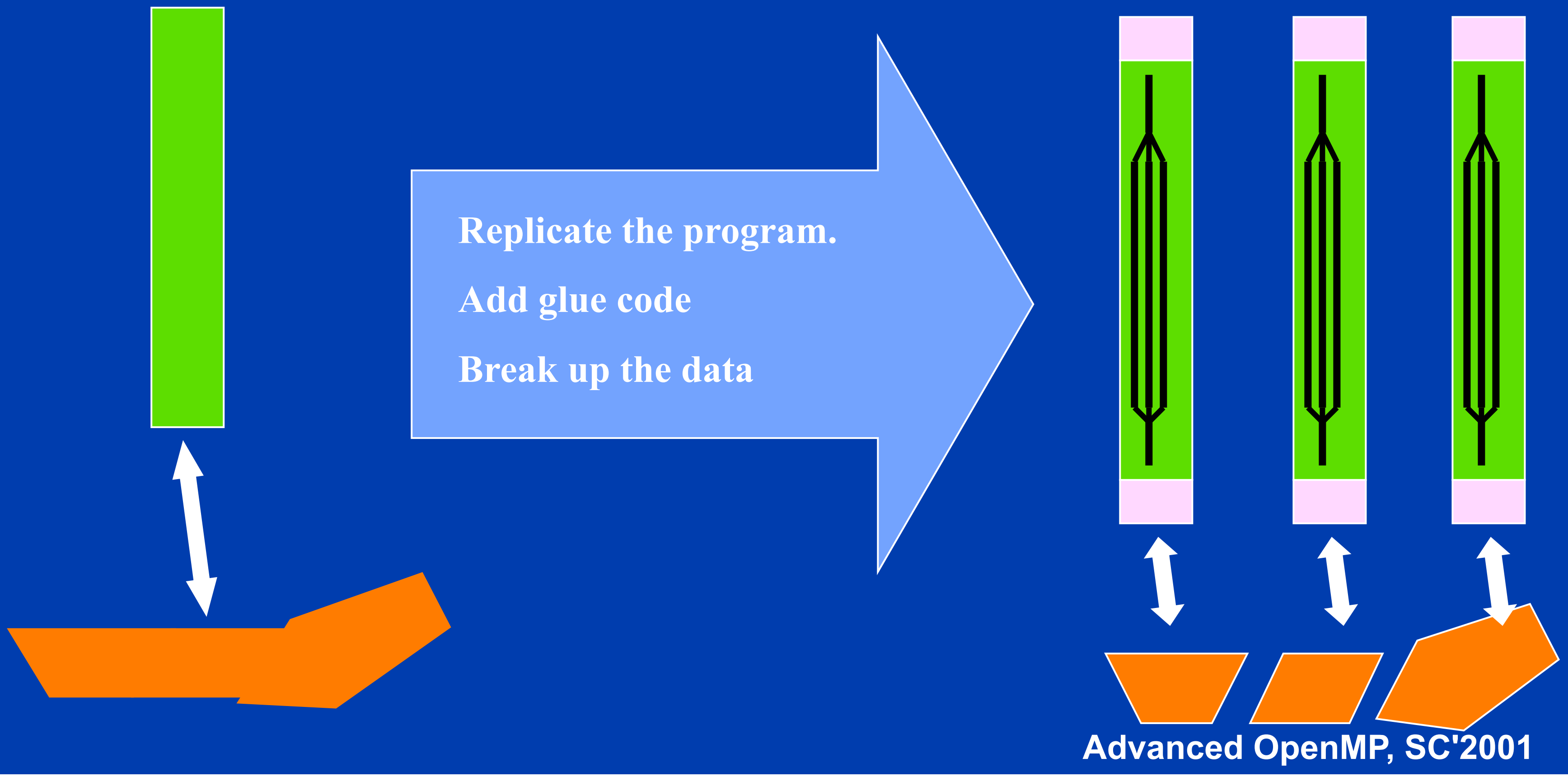
Pi program in MPI

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=myrank*my_steps; i<(myrank+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD) ;
}
```

How do people mix MPI and OpenMP?

A sequential program
working on a data set

- Create the MPI program with its data decomposition.
- Use OpenMP inside each MPI process.



Replicate the program.

Add glue code

Break up the data

Pi program with MPI and OpenMP

```
#include <mpi.h>
#include “omp.h”
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
#pragma omp parallel do
        for (i=myrank*my_steps; i<(myrank+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD) ;
}
```

Get the MPI part done first, then add OpenMP pragma where it makes sense to do so

Mixing OpenMP and MPI

Let the programmer beware!

- Messages are sent to a process on a system not to a particular thread
  - Safest approach -- only do MPI inside serial regions.
  - … or, do them inside MASTER constructs.
  - … or, do them inside SINGLE or CRITICAL
    - But this only works if your MPI is really thread safe!
- Environment variables are not propagated by mpirun. You' ll need to broadcast OpenMP parameters and set them with the library routines.

Mixing OpenMP and MPI

- OpenMP and MPI coexist by default:
  - MPI will distribute work across processes, and these processes may be threaded.
  - OpenMP will create multiple threads to run a job on a single system.
- But be careful … it can get tricky:
  - Messages are sent to a process on a system not to a particular thread.
  - Make sure you implementation of MPI is threadsafe.
  - Mpirun doesn’t distribute environment variables so your OpenMP program shouldn’t depend on them.

Dangerous Mixing of MPI and OpenMP

- The following will work on some MPI implementations, but may fail for others: MPI libraries are not always thread safe.

```
MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;
#pragma omp parallel
{
    int tag, swap_neigh, stat, omp_id = omp_thread_num();
    long buffer [BUFF_SIZE], incoming [BUFF_SIZE];
    big_ugly_calc1(omp_id, mpi_id, buffer);
                                                                // Finds MPI id and tag
so
    neighbor(omp_id, mpi_id, &swap_neigh, &tag);  // messages don’t
conflict

    MPI_Send (buffer,   BUFF_SIZE, MPI_LONG, swap_neigh,
                tag, MPI_COMM_WORLD);
    MPI_Recv (incoming, buffer_count, MPI_LONG, swap_neigh,
                tag,  MPI_COMM_WORLD, &stat);

    big_ugly_calc2(omp_id, mpi_id, incoming, buffer);
#pragma critical
   consume(buffer, omp_id, mpi_id);
```

Messages and threads

- Keep message passing and threaded sections of your program separate:
  - Setup message passing outside OpenMP regions
  - Surround with appropriate directives (e.g. critical section or master)
  - For certain applications depending on how it is designed it may not matter which thread handles a message.
    - Beware of race conditions though if two threads are probing on the same message and then racing to receive it.

Safe Mixing of MPI and OpenMP

Put MPI in sequential regions

```
MPI_Init(&argc, &argv) ;      MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;

// a whole bunch of initializations

#pragma omp parallel for
for (I=0;I<N;I++) {
     U[I] =  big_calc(I);
}

     MPI_Send (U,   BUFF_SIZE, MPI_DOUBLE, swap_neigh,
                    tag, MPI_COMM_WORLD);
     MPI_Recv (incoming, buffer_count, MPI_DOUBLE, swap_neigh,
                    tag,  MPI_COMM_WORLD, &stat);

#pragma omp parallel for
for (I=0;I<N;I++) {
     U[I] =  other_big_calc(I, incoming);
}

consume(U, mpi_id);
```

Safe Mixing of MPI and OpenMP

Protect MPI calls inside a parallel region

```
MPI_Init(&argc, &argv) ;      MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;

// a whole bunch of initializations

#pragma omp parallel
{
#pragma omp for
    for (I=0;I<N;I++)    U[I] =  big_calc(I);

#pragma master
{
     MPI_Send (U,   BUFF_SIZE, MPI_DOUBLE, neigh, tag,  MPI_COMM_WORLD);
     MPI_Recv (incoming, count, MPI_DOUBLE, neigh,  tag,  MPI_COMM_WORLD,
                                                                   &stat);
}
#pragma omp barrier
#pragma omp for
    for (I=0;I<N;I++)   U[I] =  other_big_calc(I, incoming);

#pragma omp master
    consume(U, mpi_id);
}
```

MPI and Environment Variables

- Environment variables are not propagated by mpirun, so you may need to explicitly set the requested number of threads with OMP_NUM_THREADS().

Agenda

- Summary of OpenMP basics
- OpenMP: The more subtle/advanced stuff
- OpenMP case studies
- Automatic parallelism and tools support
- Mixing OpenMP and MPI
- ➡ The future of OpenMP
  - Updating C/C++
  - Longer Term issues

Updating OpenMP for C/C++

- Two step process to update C/C++
  - OpenMP 2.0: Bring the 1.0 specification up to date:
    - Line up OpenMP C/C++ with OpenMP Fortran 2.0
    - Line up OpenMP C/C++ with C99.
  - OpenMP 3.0: Add new functionality to extend the scope and value of OpenMP.
- Target is to have a public review draft of OpenMP 2.0 C/C++ at SC'2001.

OpenMP 2.0 for C/C++

Line up with OpenMP 2.0 for Fortran

- Specification of the number of threads with the NUM_THREADS clause.
- Broadcast a value with the COPYPRIVATE clause.
- Extension to THREADPRIVATE.
- Extension to CRITICAL.
- New timing routines.
- Lock functions can be used in parallel regions.

NUM_THREADS Clause

- Used with a parallel construct to request number of threads used in the parallel region.
    - supersedes the omp_set_num_threads library function, and the OMP_NUM_THREADS environment variable.

```
#include <omp.h>
main () {
...
omp_set_dynamic(1);
...
#pragma omp parallel for num_threads(10)
   for (i=0; i<10; i++)
      {
         ...
      }
}
```

COPYPRIVATE

- Broadcast a private variable from one member of a team to the other members.
- Can only be used in combination with SINGLE

```
float x, y;
#pragma omp threadprivate(x, y)

void init(float a, float b)
{
   #pragma omp single copyprivate(a,b,x,y)
   {
       get_values(a,b,x,y);
    }
}
```

Extension to THREADPRIVATE

- OpenMP Fortran 2.0 allows SAVE'd variables to be made THREADPRIVATE.
- The corresponding functionality in OpenMP C/C++ is for function local static variables to be made THREADPRIVATE.

```
int sub()
{
  static int gamma = 0;
  static int counter = 0;
#pragma omp threadprivate(counter)
  gamma++;
  counter++:
  return(gamma);
}
```

Extension to CRITICAL Construct

- In OpenMP C/C++ 1.0, critical regions can not contain worksharing constructs.
- This is allowed in OpenMP C/C++ 2.0, as long as the worksharing constructs do not bind to the same parallel region as the critical construct.

```
void f() {
int i = 1;
#pragma omp parallel sections
  {
#pragma omp section
    {
#pragma omp critical (name)
      {
#pragma omp parallel
        {
#pragma omp single
           {
              i++;
           } } } } } }
```

Timing Routines

- Two functions have been added in order to support a portable wall-clock timer:
  - double omp_get_wtime(void);
    returns elapsed wall-clock time
  - double omp_get_wtick(void);
    returns seconds between successive clock ticks.

```
double start;
double end;
start = omp_get_wtime();
… work to be timed …
end = omp_get_wtime();
printf(“Work took %f sec. Time.\n”, end-start);
```

Thread-safe Lock Functions

- OpenMP 2.0 C/C++ lets users initialise locks in a parallel region.

```
#include <omp.h>

omp_lock_t *new_lock()
{
  omp_lock_t *lock_ptr;
#pragma omp single copyprivate(lock_ptr)
  {
    lock_ptr = (omp_lock_t *)
                  malloc(sizeof(omp_lock_t));
    omp_init_lock( lock_ptr );
  }
return lock_ptr;
}
```

Reprivatization

- Private variables can be marked private again in a nested directive. They do not have to be shared in the enclosing parallel region anymore.
- This does not apply to the FIRSTPRIVATE and LASTPRIVATE directives.

```
int a;
...
#pragma omp parallel private(a)
{
 ...
#pragma omp parallel for private(a)
for (i=0; i<n; i++) {
   ...
   }
}
```

OpenMP 2.0 for C/C++

Line up with C99

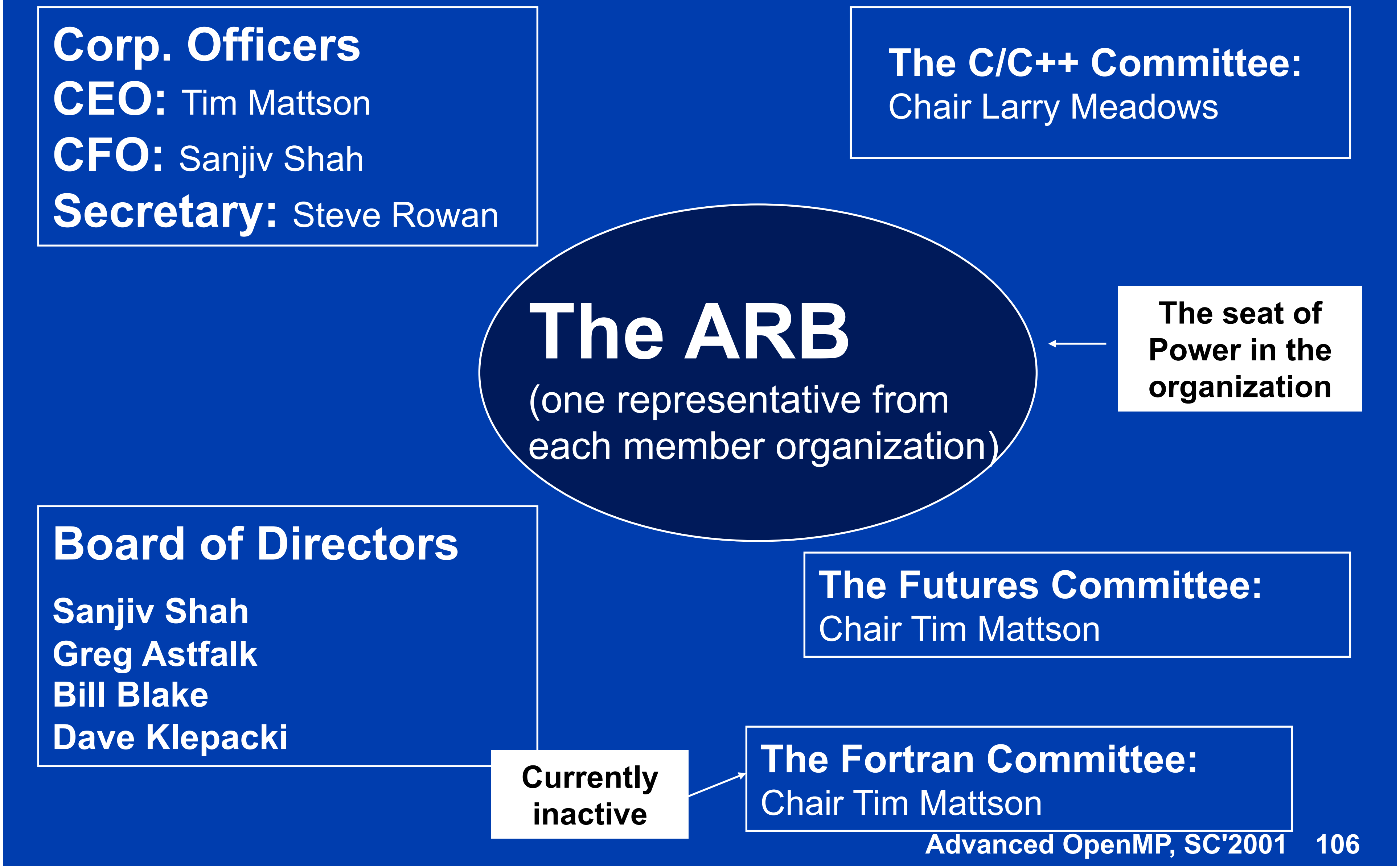
- C99 variable length arrays are complete types, thus they can be specified anywhere complete types are allowed.
- Examples are the private, firstprivate, and lastprivate clauses.

```
void f(int m, int C[m][m])
{
double v1[m];
...
#pragma omp parallel firstprivate(C, v1)
...
}
```

Agenda

- Summary of OpenMP basics
- OpenMP: The more subtle/advanced stuff
- OpenMP case studies
- Automatic parallelism and tools support
- Mixing OpenMP and MPI
- The future of OpenMP
  - Updating C/C++
  - ➡ Longer Term issues

OpenMP Organization



Corp. Officers
CEO: Tim Mattson
CFO: Sanjiv Shah
Secretary: Steve Rowan

The C/C++ Committee:
Chair Larry Meadows

The ARB
(one representative from each member organization)

The seat of Power in the organization

Board of Directors

Sanjiv Shah
Greg Astfalk
Bill Blake
Dave Klepacki

The Futures Committee:
Chair Tim Mattson

Currently inactive

The Fortran Committee:
Chair Tim Mattson

OpenMP

I’m worried about OpenMP

- The ARB is below critical mass.
- We are largely restricted to supercomputing.
  - I want general purpose programmers to use OpenMP. Bring on the game developers.
- Can we really “make a difference” if all we do is worry about programming shared memory computers?
  - To have a sustained impact, maybe we need to broaden our agenda to more general programming problems.
- OpenMP isn’t modular enough – it doesn’t work well with other technologies.

OpenMP ARB membership

- Due to acquisitions and changing business climate, the number of *officially distinct* ARB members is shrinking.
  - KAI acquired by Intel.
  - Compaq's compiler group joining Intel.
  - Compaq merging with HP.
  - Cray sold to Terra and dropped out of OpenMP ARB.
- We need fresh blood. cOMPunity is an exciting addition, but it would be nice to have more.

Bring more programmers into OpenMP:
Tools for OpenMP

- OpenMP is an explicit model that works closely with the compiler.
- OpenMP is conceptually well oriented to support a wide range of tools.
  - But other then KAI tools (which aren't available everywhere) there are no portable tools to work with OpenMP.
- Do we need standard Tool interfaces to make it easier for vendors and researchers to create tools?
  - We are currently looking into this on the futures committee.

Check out the Mohr, Malony et. al. paper at EWOMP' 2001

Bring more programmers into OpenMP:
Move beyond array driven algorithms

- OpenMP workshare constructs currently support:
  - iterative algorithms (omp for).
  - static non-iterative algorithms (omp sections).
- But we don’t support
  - Dynamic non-iterative algorithms?
  - Recursive algorithms?

We are looking very closely at the task queue proposal from KAI.

OpenMP Work queues

OpenMP can’t deal with a simple pointer following loop

```
nodeptr list, p;

for (p=list; p!=NULL; p=p->next)
        process(p->data);
```

KAI has proposed (and implemented) a taskq constuct to deal with this case:

```
nodeptr list, p;

#pragma omp parallel taskq
for (p=list; p!=NULL; p=p->next)
#pragma omp task
        process(p->data);
```

We need an independent evaluation of this technology

Reference: Shah, Haab, Petersen and Throop, EWOMP’1999 paper.

How should we move OpenMP beyond SMP?

- OpenMP is inherently an SMP model, but all shared memory vendors build NUMA and DVSM machines.
- What should we do?
  - Add HPF-like data distribution.
  - Work with thread affinity, clever page migration and a smart OS.
  - Give up?

OpenMP must be more modular

- Define how OpenMP Interfaces to “other stuff”:
  - How can an OpenMP program work with components implemented with OpenMP?
  - How can OpenMP work with other thread environments?
- Support library writers:
  - OpenMP needs an analog to MPI’s contexts.

We don’t have any solid proposals on the table to deal with these problems.

The role of academic research

- We need reference implementations for any new feature added to OpenMP.
  - OpenMP's evolution depends on good academic research on new API features.
- We need a good, community, open source OpenMP compiler for academics to try-out new API enhancements.
  - Any suggestions?

OpenMP will go nowhere without help from research organizations

Summary

- OpenMP is:
  - A great way to write parallel code for shared memory machines.
  - A very simple approach to parallel programming.
  - Your gateway to special, painful errors (race conditions).
- OpenMP impacts clusters:
  - Mixing MPI and OpenMP.
  - Distributed shared memory.

Reference Material on OpenMP*

OpenMP Homepage www.openmp.org:

The primary source of information about OpenMP and its development.

Books:

Parallel programming in OpenMP, Chandra, Rohit, San Francisco, Calif. : Morgan Kaufmann ; London : Harcourt, 2000, ISBN: 1558606718

OpenMP Workshops:

WOMPAT: Workshop on OpenMP Applications and Tools
    WOMPAT 2000: www.cs.uh.edu/wompat2000/
    WOMPAT 2001: www.ece.purdue.edu/~eigenman/wompat2001/
        Papers published in Lecture Notes in Computer Science #2104
EWOMP: European Workshop on OpenMP
    EWOMP 2000: www.epcc.ed.ac.uk/ewomp2000/
    EWOMP 2001: www.ac.upc.ed/ewomp2001/, held in conjunction with PACT 2001

WOMPEI: International Workshop on OpenMP, Japan
    WOMPEI 2000: research.ac.upc.jp/wompei/, held in conjunction with ISHPC 2000
        Papers published in Lecture Notes in Computer Science, #1940

* Third party trademarks and names are the property of their respective owners.

OpenMP Homepage www.openmp.org:

Corbalan J, Labarta J. Improving processor allocation through run-time measured efficiency. Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001. IEEE Comput. Soc. 2001, pp.6 pp.. Los Alamitos, CA, USA.

Saito T, Abe A, Takayama K. Benchmark of parallelization methods for unstructured shock capturing code. Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001. IEEE Comput. Soc. 2001, pp.8 pp.. Los Alamitos, CA, USA.

Mattson TG. High performance computing at Intel: the OSCAR software solution stack for cluster computing. Proceedings First IEEE/ ACM International Symposium on Cluster Computing and the Grid. IEEE Comput. Soc. 2001, pp.22-5. Los Alamitos, CA, USA.

Mattson, T.G. An Introduction to OpenMP 2.0, Proceedings 3rd International Symposium on High Performance Computing, Lecture Notes in Computer Science, Number 1940, 2000 pp. 384-390, Tokyo Japan.

Scherer A, Gross T, Zwaenepoel W. Adaptive parallelism for OpenMP task parallel programs. Languages, Compilers, and Run-Time Systems for Scalable Computers. 5th International Workshop, LCR 2000. Lecture Notes in Computer Science Vol.1915 . Springer-Verlag. 2000, pp.113-27. Berlin, Germany.

Tanaka Y, Taura K, Sato M, Yonezawa A. Performance evaluation of OpenMP applications with nested parallelism. Languages, Compilers, and Run-Time Systems for Scalable Computers. 5th International Workshop, LCR 2000. Selected Papers (Lecture Notes in Computer Science Vol.1915). Springer-Verlag. 2000, pp.100-12. Berlin, Germany.

Nikolopoulos DS, Papatheodorou TS, Polychronopoulos CD, Labarta J, Ayguade E. UPMLIB: a runtime system for tuning the memory performance of OpenMP programs on scalable shared-memory multiprocessors. Languages, Compilers, and Run-Time Systems for Scalable Computers. 5th International Workshop, LCR 2000. Selected Papers (Lecture Notes in Computer Science Vol.1915). Springer-Verlag. 2000, pp.85-99. Berlin, Germany.

Gottlieb S, Tamhankar S. Benchmarking MILC code with OpenMP and MPI. Elsevier. Nuclear Physics B-Proceedings Supplements, vol.94, March 2001, pp.841-5. Netherlands.

Balsara DS, Norton CD. Highly parallel structured adaptive mesh refinement using parallel language-based approaches. Parallel Computing, vol.27, no.1-2, Jan. 2001, pp.37-70. Publisher: Elsevier, Netherlands.

Hoeflinger J, Alavilli P, Jackson T, Kuhn B. Producing scalable performance with OpenMP: Experiments with two CFD applications. Parallel Computing, vol.27, no.4, March 2001, pp.391-413. Publisher: Elsevier, Netherlands.

Gonzalez JA, Leon C, Piccoli F, Printista M, Roda JL, Rodriguez C, Sande F. Towards standard nested parallelism. Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting. Proceedings (Lecture Notes in Computer Science Vol.1908). Springer-Verlag. 2000, pp.96-103. Berlin, Germany.

Benkner S, Brandes T. Exploiting data locality on scalable shared memory machines with data parallel programs. Euro-Par 2000 Parallel Processing. 6th International Euro-Par Conference. Proceedings (Lecture Notes in Computer Science Vol.1900). Springer-Verlag. 2000, pp.647-57. Berlin, Germany.

Seon Wook Kim, Eigenmann R. Where does the speedup go: quantitative modeling of performance losses in shared-memory programs. Parallel Processing Letters, vol.10, no.2-3, June-Sept. 2000, pp.227-38. Publisher: World Scientific, Singapore.

Baxter R, Bowers M, Graham P, Wojcik G, Vaughan D, Mould J. An OpenMP approach to parallel solvers in PZFlex. Developments in Engineering Computational Technology. Fifth International Conference on Computational Structures Technology and the Second International Conference on Engineering Computational Technology. Civil-Comp Press. 2000, pp.241-7. Edinburgh, UK.

Chapman B, Merlin J, Pritchard D, Bodin F, Mevel Y, Sorevik T, Hill L. Program development tools for clusters of shared memory multiprocessors. Journal of Supercomputing, vol.17, no.3, Nov. 2000, pp.311-22. Publisher: Kluwer Academic Publishers, Netherlands.

Vitela JE, Hanebutte UR, Gordillo JL, Cortina LM. Comparative study of message passing and shared memory parallel programming models in neural network training. Proceedings of the High Performance Computing Symposium - HPC 2000. SCS. 2000, pp.136-41. San Diego, CA, USA.

Berrendorf R, Nieken G. Performance characteristics for OpenMP constructs on different parallel computer architectures. Concurrency Practice & Experience, vol.12, no.12, Oct. 2000, pp.1261-73. Publisher: Wiley, UK.

Hisley D, Agrawal G, Satya-Narayana P, Pollock L. Porting and performance evaluation of irregular codes using OpenMP. Concurrency Practice & Experience, vol.12, no.12, Oct. 2000, pp.1241-59. Publisher: Wiley, UK.

Shah S, Haab G, Petersen P, Throop J. Flexible control structures for parallelism in OpenMP. Concurrency Practice & Experience, vol.12, no.12, Oct. 2000, pp.1219-39. Publisher: Wiley, UK.

Gonzalez M, Ayguade E, Martorell X, Labarta J, Navarro N, Oliver J. NanosCompiler: supporting flexible multilevel parallelism exploitation in OpenMP. Concurrency Practice & Experience, vol.12, no.12, Oct. 2000, pp.1205-18. Publisher: Wiley, UK.

Brunschen C, Brorsson M. OdinMP/CCp-a portable implementation of OpenMP for C. Concurrency Practice & Experience, vol.12, no.12, Oct. 2000, pp.1193-203. Publisher: Wiley, UK.

Adhianto L, Bodin F, Chapman B, Hascoet L, Kneer A, Lancaster D, Wolton L, Wirtz M. Tools for OpenMP application development: the POST project. Concurrency Practice & Experience, vol.12, no.12, Oct. 2000, pp.1177-91. Publisher: Wiley, UK.

Kuhn B, Petersen P, O'Toole E. OpenMP versus threading in C/C++. Concurrency Practice & Experience, vol.12, no.12, Oct. 2000, pp.1165-76. Publisher: Wiley, UK.

Brieger L. HPF to OpenMP on the Origin2000: a case study. Concurrency Practice & Experience, vol.12, no.12, Oct. 2000, pp. 1147-54. Publisher: Wiley, UK.

Hadish Gebremedhin A, Manne F. Scalable parallel graph colouring algorithms. Concurrency Practice & Experience, vol.12, no.12, Oct. 2000, pp.1131-46. Publisher: Wiley, UK.

Smith L, Kent P. Development and performance of a mixed OpenMP/MPI quantum Monte Carlo code. Concurrency Practice & Experience, vol.12, no.12, Oct. 2000, pp.1121-9. Publisher: Wiley, UK.

Diederichs K. Computing in macromolecular crystallography using a parallel architecture. Journal of Applied Crystallography, vol. 33, pt.4, Aug. 2000, pp.1154-61. Publisher: Munksgaard International Booksellers & Publishers, Denmark.

Couturier R. Three different parallel experiments in numerical simulation. Technique et Science Informatiques, vol.19, no.5, May 2000, pp.625-48. Publisher: Editions Hermes, France.

Piecuch P, Landman JI. Parallelization of multi-reference coupled-cluster method. Parallel Computing, vol.26, no.7-8, July 2000, pp. 913-43. Publisher: Elsevier, Netherlands.

Hong-Soog Kim, Young-Ha Yoon, Sang-Og Na, Dong-Soo Han. ICU-PFC: an automatic parallelizing compiler. Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region. IEEE Comput. Soc. Part vol.1, 2000, pp.243-6 vol.1. Los Alamitos, CA, USA.

Alan J. Wallcraft: SPMD OpenMP versus MPI for ocean models. Concurrency - Practice and Experience 12(12): 1155-1164 (2000)

JOMPan OpenMP-like interface for Java; J. M. Bull and M. E. Kambites; Proceedings of the ACM 2000 conference on Java Grande, 2000, Pages 44 - 53.

Sosa CP, Scalmani C, Gomperts R, Frisch MJ. Ab initio quantum chemistry on a ccNUMA architecture using OpenMP. III. Parallel Computing, vol.26, no.7-8, July 2000, pp.843-56. Publisher: Elsevier, Netherlands.

Bova SW, Breshears CP, Cuicchi C, Demirbilek Z, Gabb H. Nesting OpenMP in an MPI application. Proceedings of the ISCA 12th International Conference. Parallel and Distributed Systems. ISCA. 1999, pp.566-71. Cary, NC, USA.

Gonzalez M, Serra A, Martorell X, Oliver J, Ayguade E, Labarta J, Navarro N. Applying interposition techniques for performance analysis of OPENMP parallel applications. Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000. IEEE Comput. Soc. 2000, pp.235-40. Los Alamitos, CA, USA.

Chapman B, Mehrotra P, Zima H. Enhancing OpenMP with features for locality control. Proceedings of Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology. Towards Teracomputing. World Scientific Publishing. 1999, pp.301-13. Singapore.

Cappello F, Richard O, Etiemble D. Performance of the NAS benchmarks on a cluster of SMP PCs using a parallelization of the MPI programs with OpenMP. Parallel Computing Technologies. 5th International Conference, PaCT-99. Proceedings (Lecture Notes in Computer Science Vol.1662). Springer-Verlag. 1999, pp.339-50. Berlin, Germany.

Couturier R, Chipot C. Parallel molecular dynamics using OPENMP on a shared memory machine. Computer Physics Communications, vol.124, no.1, Jan. 2000, pp.49-59. Publisher: Elsevier, Netherlands.

Bova SW, Breshearsz CP, Cuicchi CE, Demirbilek Z, Gabb HA. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. International Journal of High Performance Computing Applications, vol.14, no.1, Spring 2000, pp.49-64. Publisher: Sage Science Press, USA.

Majumdar A. Parallel performance study of Monte Carlo photon transport code on shared-, distributed-, and distributed-shared-memory architectures. Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000. IEEE Comput. Soc. 2000, pp.93-9. Los Alamitos, CA, USA.

Bettenhausen MH, Ludeking L, Smithe D, Hayes S. Progress toward a parallel MAGIC. IEEE Conference Record - Abstracts. 1999 IEEE International Conference on Plasma Science. 26th IEEE International Conference. IEEE. 1999, pp.214. Piscataway, NJ, USA.

Cappello F, Richard O, Etiemble D. Investigating the performance of two programming models for clusters of SMP PCs. Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6. IEEE Comput. Soc. 1999, pp. 349-59. Los Alamitos, CA, USA.

Giordano M, Furnari MM. HTGviz: a graphic tool for the synthesis of automatic and user-driven program parallelization in the compilation process. High Performance Computing. Second International Symposium, ISHPC'99. Proceedings. Springer-Verlag. 1999, pp.312-19. Berlin, Germany.

Saito H, Stavrakos N, Polychronopoulos C. Multithreading runtime support for loop and functional parallelism. High Performance Computing. Second International Symposium, ISHPC'99. Proceedings. Springer-Verlag. 1999, pp.133-44. Berlin, Germany.

Voss M, Eigenmann R. Dynamically adaptive parallel programs. High Performance Computing. Second International Symposium, ISHPC'99. Proceedings. Springer-Verlag. 1999, pp.109-20. Berlin, Germany.

Cappello F, Richard O. Performance characteristics of a network of commodity multiprocessors for the NAS benchmarks using a hybrid memory model. 1999 International Conference on Parallel Architectures and Compilation Techniques. IEEE Comput. Soc. 1999, pp.108-16. Los Alamitos, CA, USA.

Linden P, Chakarova R, Faxen T, Pazsit I. Neural network software for unfolding positron lifetime spectra. High-Performance Computing and Networking. 7th International Conference, HPCN Europe 1999. Proceedings. Springer-Verlag. 1999, pp.1194-8. Berlin, Germany.

Silber G-A, Darte A. The Nestor library: a tool for implementing Fortran source to source transformations. High-Performance Computing and Networking. 7th International Conference, HPCN Europe 1999. Proceedings. Springer-Verlag. 1999, pp.653-62. Berlin, Germany.

Kessler CW, Seidl H. ForkLight: a control-synchronous parallel programming language. High-Performance Computing and Networking. 7th International Conference, HPCN Europe 1999. Proceedings. Springer-Verlag. 1999, pp.525-34. Berlin, Germany.

Prins JF, Chatterjee S, Simons M. Irregular computations in Fortran-expression and implementation strategies. Scientific Programming, vol.7, no.3-4, 1999, pp.313-26. Publisher: IOS Press, Netherlands.

Adve SV, Pai VS, Ranganathan P. Recent advances in memory consistency models for hardware shared memory systems. Proceedings of the IEEE, vol.87, no.3, March 1999, pp.445-55. Publisher: IEEE, USA.

Chapman B, Mehrotra P. OpenMP and HPF: integrating two paradigms. Euro-Par'98 Parallel Processing. 4th International Euro-Par Conference. Proceedings. Springer-Verlag. 1998, pp.650-8. Berlin, Germany.

Rauchwerger L, Arzu F, Ouchi K. Standard Templates Adaptive Parallel Library (STAPL). Languages, Compilers, and Run-Time Systems for Scalable Computers. 4th International Workshop, LCR '98. Selected Papers. Springer-Verlag. 1998, pp.402-9. Berlin, Germany.

Beckmann CJ, McManus DD, Cybenko G. Horizons in scientific and distributed computing. Computing in Science & Engineering, vol.1, no.1, Jan.-Feb. 1999, pp.23-30. Publisher: IEEE Comput. Soc, USA.

Scherer A, Honghui Lu, Gross T, Zwaenepoel W. Transparent adaptive parallelism on NOWS using OpenMP. ACM. Sigplan Notices (Acm Special Interest Group on Programming Languages), vol.34, no.8, Aug. 1999, pp.96-106. USA.

Ayguade E, Martorell X, Labarta J, Gonzalez M, Navarro N. Exploiting multiple levels of parallelism in OpenMP: a case study. Proceedings of the 1999 International Conference on Parallel Processing. IEEE Comput. Soc. 1999, pp.172-80. Los Alamitos, CA, USA.

Honghui Lu, Hu YC, Zwaenepoel W. OpenMP on networks of workstations. Proceedings of ACM/IEEE SC98: 10th Anniversary. High Performance Networking and Computing Conference. IEEE Comput. Soc. 1998, pp.13 pp.. Los Alamitos, CA, USA.

Throop J. OpenMP: shared-memory parallelism from the ashes. Computer, vol.32, no.5, May 1999, pp.108-9. Publisher: IEEE Comput. Soc, USA.

Hu YC, Honghui Lu, Cox AL, Zwaenepoel W. OpenMP for networks of SMPs. Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999. IEEE Comput. Soc. 1999, pp.302-10. Los Alamitos, CA, USA.

Still CH, Langer SH, Alley WE, Zimmerman GB. Shared memory programming with OpenMP. Computers in Physics, vol.12, no.6, Nov.-Dec. 1998, pp.577-84. Publisher: AIP, USA.

Chapman B, Mehrotra P. OpenMP and HPF: integrating two paradigms. Euro-Par'98 Parallel Processing. 4th International Euro-Par Conference. Proceedings. Springer-Verlag. 1998, pp.650-8. Berlin, Germany.

Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. IEEE Computational Science & Engineering, vol.5, no.1, Jan.-March 1998, pp.46-55. Publisher: IEEE, USA.

Clark D. OpenMP: a parallel standard for the masses. IEEE Concurrency, vol.6, no.1, Jan.-March 1998, pp.10-12. Publisher: IEEE, USA.

Extra Slides

A series of parallel pi programs

Some OpenMP Commands to support Exercises

PI Program: an example

```
static long num_steps = 100000;
double step;
void main ()
{          int i;   double x, pi, sum = 0.0;

           step = 1.0/(double) num_steps;

           for (i=1;i<= num_steps; i++){
                   x = (i-0.5)*step;
                   sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
}
```

Parallel Pi Program

- Let’ s speed up the program with multiple threads.
- Consider the Win32 threads library:
  - Thread management and interaction is explicit.
  - Programmer has full control over the threads

Solution: Win32 API, PI

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
   int i, start;
  double x, sum = 0.0;


  start = *(int *) arg;
  step = 1.0/(double) num_steps;

  for (i=start;i<= num_steps; i=i+NUM_THREADS){
      x = (i-0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
  }
  EnterCriticalSection(&hUpdateMutex);
  global_sum += sum;
  LeaveCriticalSection(&hUpdateMutex);
}
```

```
void main ()
{
  double pi; int i;
  DWORD threadID;
  int threadArg[NUM_THREADS];

  for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

  InitializeCriticalSection(&hUpdateMutex);

  for (i=0; i<NUM_THREADS; i++){
            thread_handles[i] = CreateThread(0, 0,
                                  (LPTHREAD_START_ROUTINE) Pi,
                                  &threadArg[i], 0, &threadID);
}

  WaitForMultipleObjects(NUM_THREADS,
                                  thread_handles, TRUE,INFINITE);

  pi = global_sum * step;

  printf(" pi is %f \n",pi);
}
```

Doubles code size!

Solution: Keep it simple

Threads libraries:

- Pro: Programmer has control over everything
- Con: Programmer must control everything

Full control

Increased complexity

Programmers scared away

Sometimes a simple evolutionary approach is better

OpenMP PI Program:

Parallel Region example (SPMD Program)

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
void main ()
{               int i;      double x, pi, sum[NUM_THREADS] = {0.0};
                step = 1.0/(double) num_steps;
                omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{               double x;     int i, id;
                id = omp_get_thraead_num();
                for (i=id;i< num_steps; i=i+NUM_THREADS){
                              x = (i+0.5)*step;
                              sum[id] += 4.0/(1.0+x*x);
                }
}
                for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

SPMD Programs:

Each thread runs the same code with the thread ID selecting any thread specific behavior.

OpenMP PI Program:

Work sharing construct

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
void main ()
{              int i;       double x, pi, sum[NUM_THREADS] = {0.0};
               step = 1.0/(double) num_steps;
               omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{              double x;     int i, id;
               id = omp_get_thraead_num();
#pragma omp for
               for (i=id;i< num_steps; i++){
                            x = (i+0.5)*step;
                            sum[id] += 4.0/(1.0+x*x);
               }
}              for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

OpenMP PI Program:

private clause and a critical section

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{          int i;      double  x, sum, pi=0.0;
           step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (x, sum,i)
{
           id = omp_get_thread_num();
           for (i=id,sum=0.0;i< num_steps;i=i+NUM_THREADS){
                    x = (i+0.5)*step;
                    sum += 4.0/(1.0+x*x);
           }
#pragma omp critical
           pi += sum * step;
}
}
```

Note: We didn't need to create an array to hold local sums or clutter the code with explicit declarations of "x" and "sum".

OpenMP PI Program :

Parallel for with a reduction

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
void main ()
{           int i;     double x, pi, sum = 0.0;
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
            for (i=1;i<= num_steps; i++){
                      x = (i-0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

OpenMP adds 2 to 4 lines of code

MPI: Pi program

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                        MPI_COMM_WORLD) ;
}
```