

Locality-aware Load Balancing for Dynamic Irregular Computations



Sriram Krishnamoorthy

P Sadayappan

The Ohio State University

Jarek Nieplocha

Manoj Krishnan

Pacific Northwest National Laboratory

Outline

- Computational Model
- Motivation
- Dynamic load-balancing pattern
- Work-sharing API
- Implementation issues
- Conclusion

Computational Model

- Computations expressible as directed acyclic graphs
 - Node – Computation
 - Edges – Input/Output relationships
 - A node is ready for evaluation when all its children have been evaluated
- Globally addressed distributed memory system
 - ARMCI/Global Arrays support
 - Data distributed amongst the processors memories – but globally addressable
- Cost of evaluating tasks may vary
 - Task computation times
 - Varying communication times
 - Data distribution
 - Mapping of tasks to processors

Computational Model (Contd.)

□ Objective

- Minimize completion time in evaluation of tasks

□ Issues

- Load balance the computation
 - Appropriate scheduling of tasks to processes
- Minimize communication
 - Co-locate tasks with data they consume

The Tensor Contraction Engine

Addresses software development challenges in Quantum Chemistry

- Family of models in electronic structure theory, with increasing number of terms: explosive increase in code complexity

Theory	#Terms	Year
CCD	11	1978
CCSD	48	1982
CCSDT	102	1988
CCSDTQ	183	1992

- Automatic transformation from high-level specification to efficient program optimized for target computer system
- Multi-institutional collaboration (Ohio State U., U. Waterloo, Oak Ridge Natl. Lab., Pacific Northwest Natl. Lab.)

$$A3A = \frac{1}{2} (X_{ce,af} Y_{ae,cf} + X_{ce,a} Y_{ae,c} + X_{ce,af} Y_{ae,cf} + X_{ce,a} Y_{ae,c} + X_{ce,af} Y_{ae,cf} + X_{ce,a} Y_{ae,c})$$

$$X_{ce,af} = t_{ij}^{ce} t_{ij}^{af} \quad Y_{ae,cf} = \langle ab || ek \rangle \langle cb || fk \rangle$$

```
range V = 3000;
range O = 100;
```

```
index a,b,c,d,e,f : V;
index i,j,k : O;
```

```
mlimit = 10000000;
```

```
function F1(V,V,V,O);
function F2(V,V,V,O);
```

```
procedure P(in T1[O,O,V,V], in T2[O,O,V,V], out X)=
```

```
begin
  A3A == sum[ sum[F1(a,b,e,k) * F2(c,f,b,k),
    {b,k}]
    * sum[T1[i,j,c,e] * T2[i,j,a,f], {i,j}],
    {a,e,c,f}]*0.5 + ...;
```

Shared Iterator: Pattern for Dynamic Load-balancing

- ❑ Set of independent tasks that share input data
- ❑ Minimize completion time
- ❑ Commonly-used solution to the problem

Shared Iterator

- Pattern Name: Shared Iterator
- Intent
 - Dynamic load-balancing of a set of independent tasks, written using a global address-space model.
- Motivation
 - Task evaluation costs can vary widely
 - Static scheduling
 - Estimation of task evaluation costs difficult

Shared Iterator - Applicability

- ❑ Global address-space models
- ❑ Atomic operations such as *fetch-and-add*
- ❑ Computation partitioned into set of independent tasks
- ❑ Number of tasks much more than number of processes

Shared Iterator - Solution

- Partition the computation into independent tasks
 - Eg: Some loop indices might be selected as parallelized
- Tasks form an ordered list
- Initialize a global shared counter to zero
- Whenever a process does not have a task to evaluate
 - Obtaining a task index
 - atomic *fetch-and-add* of the shared counter
 - Evaluate that task

Shared Iterator (Contd.)

- Consequences
 - Good computation load balance
 - Ignores locality among tasks
- Implementation
 - Implicit vs. explicit task list
 - Task granularity
 - Efficient computation
 - Overshadow communication cost
 - Maximize parallelism
 - Scalability of shared counter updates

Shared Iterator Example – Explicit Tasks

!Global data

ga, gb, gc: global arrays;
sc: shared counter;

!Local data

tsk: Task index

```
struct {  
    int i, j, k;  
} tasks[N*N*N];
```

idx=0;

for i = 0 to N-1

for j = 0 to N-1

for k = 0 to N-1

tasks[idx].i = i

tasks[idx].j = j

tasks[idx].k = k

idx = idx + 1

```
initSharedCounter(sc)
```

!Atomic fetch-and-add

```
tsk=getNextTask(sc)
```

```
while tsk < N*N*N
```

```
    i = tasks[tsk].i
```

```
    j = tasks[tsk].j
```

```
    k = tasks[tsk].k
```

```
    c = get_value(gc, i, j)
```

```
    a = get_value(ga, i, k)
```

```
    b = get_value(gb, k, j)
```

```
    c = a*b
```

```
    update_value(gc, c, i, j)
```

```
    tsk = getNextTask(sc)
```

Shared Iterator Example – Implicit Tasks

!Global data

ga, gb, gc: global arrays;
sc: shared counter;

!Local data

tsk: Task index

```
initSharedCounter(sc)
```

```
!Atomic fetch-and-add
```

```
tsk=getNextTask(sc)
```

```
for i = 0 to N-1
```

```
  for j = 0 to N-1
```

```
    for k = 0 to N-1
```

```
      if  $i*N*N + j*N + k$  == tsk
```

```
        !My task
```

```
          c = get_value(gc, i, j)
```

```
          a = get_value(ga, i, k)
```

```
          b = get_value(gb, k, j)
```

```
          c = a*b
```

```
          update_value(gc, c, i, j)
```

```
          tsk = getNextTask(sc)
```

Known Uses

- Lennard Jones using force decomposition
 - Balancing computation of force contributions
- Parallel Dense Matrix Multiplication
 - Balance computation of tiles
- So Hirata's Tensor Contraction Engine
 - Sequence of tensor contractions
 - Each treated as a parallel block-sparse matrix multiply
- Lotrich et al.?

Related Patterns

- Iterator
 - Iterate through tasks
- Master-Worker
 - One mechanism for dynamic scheduling
 - Implies some program-structure
 - Pattern presented here leads to an API

Work-sharing Construct

- ❑ Library support for Shared Iterator
- ❑ Extend the pattern to DAGs
- ❑ Exploit locality between tasks

Work-sharing API

```
work_pool wp;
```

```
work_element we {  
    fn_handle,  
    input_ga_list, input_ga_range_list, input_we_list,  
    result_dims,  
    output_ga, output_ga_range  
}
```

```
wp = create_work_pool();
```

```
add_work_element(wp, we);
```

```
seal_work_element(wp);
```

```
process_work_pool(wp);
```

Efficient Work-sharing Implementation

- Locality-aware scheduling algorithms
 - Combination of static and dynamic approaches
- Global structural support to enable tasks
 - When all children of a node have been evaluated
- Computation-communication overlap
- Disk I/O
 - Virtual memory approach
 - Explicit I/O invocations

Related Work/Patterns

- Static scheduling strategies
- Cilk/CHARM++
- Start-time optimizations
 - Inspector-executor model
- SIAL/SIP - Lotrich et al.
- Berna Massingill

Conclusion

- ❑ Identified a design pattern for dynamic load-balancing of independent tasks
- ❑ Provide library support for the pattern
 - Work-sharing construct
 - Handling locality and DAGs
- ❑ Issues to be dealt with in implementing the work-sharing construct.