Enhancing Self-Scheduling Algorithms via Synchronization and Weighting

Florina M. Ciorba*

joint work with

I. Riakiotakis*, T. Andronikos†, G. Papakonstantinou* and A. T. Chronopoulos‡

* *National Technical University of Athens, Greece*
† *Ionian University, Corfu, Greece*
‡ *University of Texas at San Antonio, USA*

2nd Scheduling in Aussois Workshop, France
May 20, 2008

Outline

Outline

Outline

Problem Statement

Definition (Task Scheduling)

Given a set of tasks of a parallel computation, determine how the tasks can be assigned (both in space and time) to processing resources (scheduled on them) to satisfy certain optimality criteria.

Challenges

- minimizing execution time
- minimizing inter-processor communication
- load balancing the tasks among processors
- handling and/or recovering from failures
- meeting deadlines
- a combination of the above

Outline

Addressing the Problem of Task Scheduling

How easy/difficult is it to schedule tasks?

- ☹ Scheduling dependent tasks onto a set of homogeneous resources, considering interprocessor communication, and aiming to minimize the total execution time is NP-complete.
- ☹ The same holds for heterogeneous systems.

Make realistic assumptions regarding:

- processor heterogeneity, communication link heterogeneity, irregularity of interconnection networks, non-dedicated platforms

Solutions:

- ☹ Optimal - there are no polynomial time optimal solutions
- Heuristic methods - various (static/dynamic) scheduling heuristics have been proposed

Outline

Static vs Dynamic Scheduling

Definition (Static Scheduling)

Static scheduling involves assigning the tasks to processors before the execution of the problem, in a non-preemptive fashion. The application characteristics are known before program execution and the state of the target system does not change during the parallel execution.

Pros ☺

- Easy to design and program
- Very low scheduling overhead

Cons ☹

- Cannot cope with applications with irregular tasks
- Cause high load imbalance on heterogeneous systems

Static vs Dynamic Scheduling

Definition (Dynamic Scheduling)

In dynamic scheduling, only a few assumptions about the parallel application or the target system can be made before execution, and thus, scheduling decisions have to be made ***on-the-fly****.*

Pros ☺

- Offer good load balance on heterogeneous systems
- Can tackle applications with irregular tasks as well

Cons ☹

- Higher scheduling overhead than static methods
- Harder to design and program

Dynamic Task Scheduling

What are the goals of dynamic scheduling?

To minimize the program completion time and minimize the scheduling overhead which constitutes a significant portion of the cost paid for running the dynamic scheduler.

Why do we need dynamic scheduling?

Dynamic scheduling is necessary when static scheduling may result in a highly imbalanced distribution of work among processors or when the inter-tasks dependencies are dynamic (e.g. due to changing system's behavior or changing application's behavior), thus precluding a static scheduling approach.

Outline

What has been done so far?

- Numerous static algorithms devised for either DOALL or DOACROSS loops on homogeneous and/or heterogeneous systems
- Numerous dynamic algorithms devised for DOALL loops on homogeneous and/or heterogeneous systems

What is missing?

Dynamic scheduling and load balancing algorithms for DOACROSS loops on heterogeneous systems

Outline

Why deal with dynamic load balancing algorithms?

Motivation:

- Existing dynamic load balancing algorithms (self-scheduling) can not cope with task dependencies, because they lack inter-slave communication
- If dynamic load balancing algorithms are applied to DOACROSS loops, in their original form, they yield a very slow/serial execution
- Static algorithms are not always efficient on heterogeneous systems

What is needed?

- The current dynamic load balancing algorithms (self-scheduling) need something to enable them to handle DOACROSS loops and something else to enable them to be efficient on heterogeneous systems

Why deal with dynamic load-balancing algorithms?

Contributions:

- A synchronization mechanism (the 'something') based on an extended master-slave model that provides inter-slave communication
- A weighting mechanism (the 'something else') that adjusts the amount of work assigned to a processor according to its performance

Outline

Outline

DOACROSS loops - algorithmic model

```
for  (i_1 = l_1; i_1 <= u_1; i_1++)
     for  (i_2 = l_2; i_2 <= u_2; i_2++)
     ...
          for  (i_n = l_n; i_n <= u_n; i_n++)
               S_1(I);
               ...
               S_k(I);
          endfor
     ...
     endfor
endfor
```

- $J = \{\mathbf{I} \in \mathbb{N}^n | l_r \leq i_r \leq u_r, 1 \leq r \leq n\}$ - the Cartesian n-dimensional index space of a loop of depth n
- $|J| = \prod_{i=1}^{n}(u_i - l_i + 1)$ - the cardinality of J
- $S_i(\mathbf{I})$ - general program statements of the loop body
- $DS = \{\tilde{\mathbf{d}}_1, \ldots, \tilde{\mathbf{d}}_p\}$, $p \geq n$ - the set of dependence vectors
- By definition $\tilde{\mathbf{d}}_j > \mathbf{0}$, where $\mathbf{0} = (0, \ldots, 0)$ and $>$ is the *lexicographic* ordering
- $\mathbf{L} = (l_1, \ldots, l_n)$ - the initial point of J
- $\mathbf{U} = (u_1, \ldots, u_n)$ - the terminal point of J

Graphical representations of DOACROSS loops using Cartesian spaces

Cartesian Spaces - the points have coordinates and represent tasks and the directed vectors represent the dependencies among the tasks (e.g. precedence)

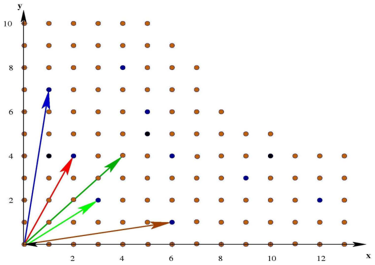
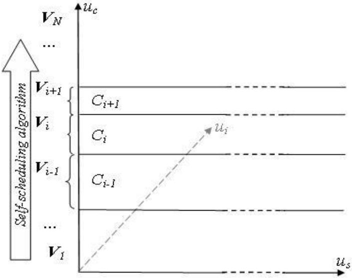


Figure: Cartesian representation of tasks and dependencies

Outline

Partitioning the Index Space with Self-Scheduling Algorithms



- u_c - scheduling dimension (1D partitioning)
- $P_1, \ldots, P_m$ - slave processors; P_0 - master processor
- N - the number of scheduling steps (the total number of chunks)
- C_i - chunk size at the i-th scheduling step
- V_i - the projection of C_i along scheduling dimension u_c
- $C_i = V_i \times \frac{\prod_{j=1}^{n} u_j}{u_c}$

- VP_k - virtual computing power of slave P_k (delivered speed)
- q_k - number of processes in the run-queue of slave P_k
- $A_k = \lfloor \frac{VP_k}{q_k} \rfloor$ - available computing power of slave P_k (delivered speed)
- $A = \sum_{i=1}^{m} A_k$ - total available computing power of the system

Overview of Self-Scheduling Algorithms for DOALL Loops

Obs. They use a simple master-slave model

PSS - Pure Self-Scheduling, $C_i = 1$

CSS [Kruskal and Weiss, 1985] - Chunk Self-Scheduling, $C_i = \text{constant} > 1$

GSS [Polychronopoulos and Kuck, 1987] – Guided Self-Scheduling, $C_i = R_i/m$, where R_i is the number of remaining iterations

Overview of Self-Scheduling Algorithms for DOALL Loops

FSS [Hummel et al, 1992] – Factoring Self-Scheduling, assigns batches of equal chunks. $C_i = \lceil \frac{R_i}{\alpha * m} \rceil$ and $R_{i+1} = R_i - (m \times C_i)$, where the parameter α is computed (by a probability distribution) or is sub-optimally chosen $\alpha = 2$.

TSS [Tzen and Ni, 1993] - Trapezoid Self-Scheduling, $C_i = C_{i-1} - D$, where D decrement, the first chunk is $F = \frac{|J|}{2m}$ and the last chunk is $L = 1$

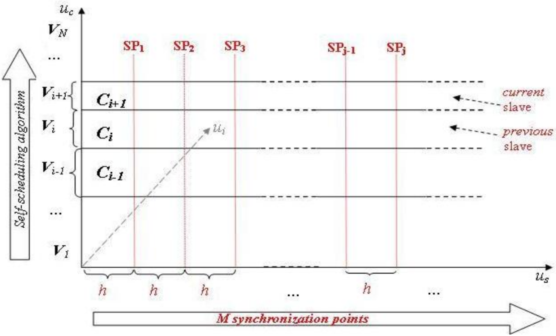
DTSS [Chronopoulos et al, 2001] - Distributed TSS, $C_i = A_k \times (F - D \times (S_{k-1} + (A_k - 1)/2))$, where: $S_{k-1} = A_1 + \ldots + A_{k-1}$, the first chunk is $F = \frac{|J|}{2A}$ and the last chunk is $L = 1$

Overview of Self-Scheduling Algorithms for DOALL Loops

Algorithm	Pros ☺	Cons ☹	Heterogeneity?
PSS	good load bal.	excessive sch. & comm. ovhd	no
CSS	low sch. ovhd.	large chunks ⇒ load imbalance small chunks ⇒ excessive comm. ovhd.	no
GSS	low sch. ovhd. large chunks first ⇒ reduced comm. small chunks last ⇒ good load bal.	⇒ may cause load imbalance	no
FSS	improves on GSS low sch. ovhd. few chunk adaptations (batches)	difficult to determine the optimal parameters for batching	no
TSS	low sch. ovhd. (constant decrement) improves on GSS for irregular tasks	difficult to determine the optimal parameters (F, L, D)	no
DTSS	improves on TSS by assigning chunks to processors according to their delivered speed	difficult to determine the optimal parameters (F, L, D)	yes

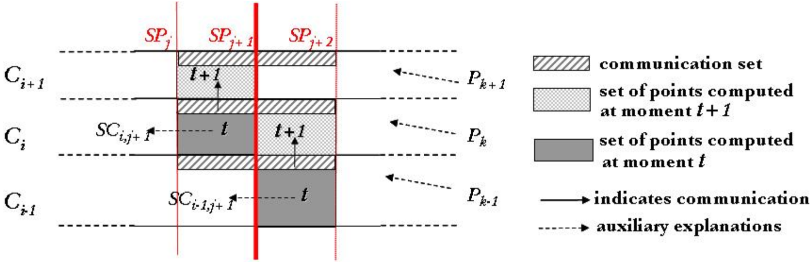
Outline

Self-Scheduling for DOACROSS loops with Synchronization Points



- Chunks are formed along the scheduling dimension, u_c
- *SPs* are inserted along the synchronization dimension, u_s

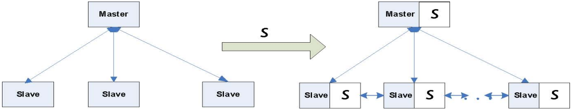
The Inter-slave Communication Scheme



- C_{i-1} is assigned to P_{k-1}, C_i assigned to P_k and C_{i+1} to P_{k+1}
- When P_k reaches SP_{j+1}, it sends to P_{k+1} only the data P_{k+1} requires (i.e., those iterations imposed by the existing dependence vectors)
- Next, P_k receives from P_{k-1} the data required for the current computation

Obs. Slaves do not reach a *SP* at the same time, which leads to a **pipelined execution**

The Synchronization Mechanism $\mathscr{S}$



(a) Classic master-slave model

(b) Master-slave model with the stand alone synchronization component and inter-slaves communication links

- Enables self-scheduling algorithms to handle DOACROSS loops
- Provides:
  - The synchronization interval h along u_s: $h = \frac{U_s}{M}$
  - A framework for inter-slave communication (presented earlier)

Observations:

1. $\mathscr{S}$ is completely independent of the self-scheduling algorithm and does not enhance the load balancing capability of the algorithm
2. The synchronization overhead is compensated by the increase of parallelism $\Rightarrow$ overall performance improvement

The Synchronization Mechanism $\mathscr{S}$

Master

While there are unassigned chunks
{
1. Receive request from P_k
2. Calculate C_i according to $\mathscr{A}$
3. Serve request
}

$\mathscr{S}$-Master

While there are unassigned chunks
{
1. Receive request from P_k
2. Calculate C_i according to $\mathscr{A}$

3. Make P_k - current slave
4. Make P_{k-1} - previous slave
5. Send P_{k-1} the rank of P_k
6. Send P_k the rank of P_{k-1}

7. and serve request
}

Slave P_k

1. Make new request to Master
2. If request served
{
Compute chunk
}
3. Go to step 1

$\mathscr{S}$-Slave P_k

1. Make new request to Master
2. If request served
{
Receive partial results from P_{k-1}
Compute chunk
Send partial results to P_{k+1}
}
3. Go to step 1

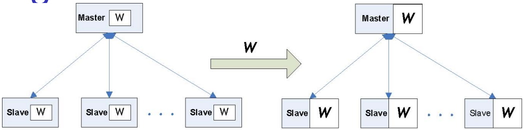
$\mathscr{S}$ adds 3 components to the original algorithm $\mathscr{A}$:

1. transaction accounting (master)
2. receive part (slave)
3. transmit part (slave)

h is determined empirically or selected by the user and must be a trade-off between synchronization overhead and parallelism

Outline

The Weighting Mechanism $\mathscr{W}$



(a) Master-slave model with embedded work-weighting

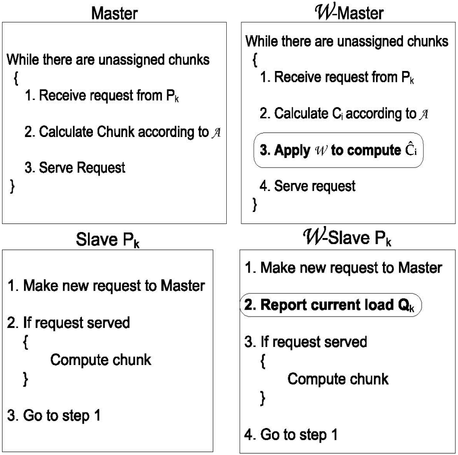
(b) Master-slave model with the stand alone weighting component

- Enables self-scheduling algorithms to handle load variations and system heterogeneity
- Adjusts the amount of work (chunk size) given by the original algorithm $\mathscr{A}$ according to the current load of a processor and its nominal computational power

Observations:

1. $\mathscr{W}$ is completely independent of the self-scheduling algorithm and can be used alone for DOALL loops
2. The weighting overhead is insignificant (a $\star$ and a $/$ operation)
3. On a dedicated homogeneous system, $\mathscr{W}$ does not improve the performance and could be omitted

The Weighting Mechanism $\mathscr{W}$



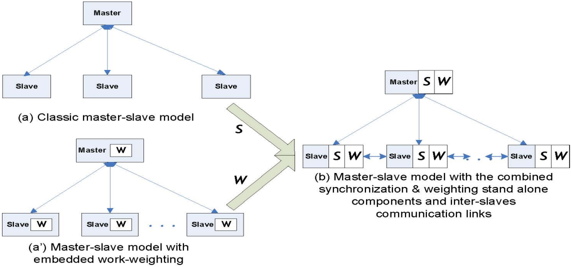
$\mathscr{W}$ adds 2 components to the original algorithm $\mathscr{A}$:

1. chunk weighting (master)
2. run-queue monitoring (slave)

$\mathscr{W}$ calculates the chunk $\widehat{C}_i$ assigned to P_k as follows: $\widehat{C}_i = C_i \times \frac{VP_k}{q_k}$, where C_i is the chunk size given by the original self-scheduling algorithm $\mathscr{A}$.

Outline

The Combined $\mathscr{S}\mathscr{W}$ Mechanisms



- $\mathscr{S}\mathscr{W}$ enable self-scheduling algorithms to handle DOACROSS loops on heterogeneous systems with load variations
- Synchronization points are introduced and chunks are weighted

Observations:

1. Since $\mathscr{S}$ does not provide any load balancing, it is most advantageous to use $\mathscr{W}$ to achieve it
2. The synchronization & weighting overheads are compensated by the performance gain

The Combined $\mathscr{S}\mathscr{W}$ Mechanisms

Master

While there are unassigned chunks
{
1. Receive request from P_k
2. Calculate C_i according to $\mathscr{A}$
3. Serve request
}

$\mathscr{S}\mathscr{W}$-Master

While there are unassigned chunks
{
1. Receive request from P_k
2. Calculate C_i according to $\mathscr{A}$
3. Apply $\mathscr{W}$ to compute $\hat{C}_i$
4. Make P_k - current slave
5. Make P_{k-1} - previous slave
6. Send P_{k-1} the rank of P_k
7. Send P_k the rank of P_{k-1}
8. Serve request
}

Slave P_k

1. Make new request to Master
2. If request served
{
Compute chunk
}
3. Go to step 1

$\mathscr{S}\mathscr{W}$-Slave P_k

1. Make new request to Master
2. Report current load Q_k
3. If request served
{
Receive partial results from P_{k-1}
Compute chunk
Send partial results to P_{k+1}
}
4. Go to step 1

$\mathscr{S}\mathscr{W}$ add 5 (3+2) components to the original algorithm $\mathscr{A}$:

1. chunk weighting (master)
2. transaction accounting (master)
3. run-queue monitoring (slave)
4. receive part (slave)
5. transmit part (slave)

Outline

Experimental Setup

- The algorithms are implemented in C and C++
- MPI is used for master-slave and inter-slave communication
- The heterogeneous system consists of 13 nodes (1 master and 12 slaves):
  - 7 twins: Intel Pentiums III, 800 MHz with 256MB RAM, assumed to have $VP_k = 1$ (one of them is the master)
  - 6 kids: Intel Pentiums III, 500 MHz with 512MB RAM , assumed to have $VP_k = 0.8$
- Interconnection network is Fast Ethernet, at 100Mbit/sec
- Non-dedicated system: at the beginning of program's execution, a resource expensive process is started on some of the slaves, halving their A_k
- Machinefile: twin1 (master),twin2, kid1, twin3, kid2, twin4, kid3, twin5, kid4, twin6, kid5, twin7, kid6
- In all cases, the kids were overloaded

Experimental Setup

- Three series of experiments on the non-dedicated system, for m = 4,6,8,10,12 slaves:

Experiment 1 for the synchronization mechanism $\mathscr{S}$
Experiment 2 for the weighting mechanism $\mathscr{W}$
Experiment 3 for the combined mechanisms $\mathscr{S}\mathscr{W}$

- Two real-life applications: Floyd-Steinberg (regular DOACROSS), and Mandelbrot (irregular DOALL)
  (*Similar results for Hydro – in* [Ciorba et al, 2008]
- Reported results are averages of 10 runs for each case
- The chunk size for CSS was: $C_i = \frac{U_c}{2 \times m}$
- The number of synchronization points was: $M = 3 \times m$
- Lower and upper thresholds for the chunk sizes (table below)
- 3 problem sizes - some analyzed here, some in [Ciorba et al, 2008]

Problem size	**small**	**medium**	**large**
Floyd-Steinberg	5000×15000	10000×15000	15000×15000
upper/lower threshold	500/10	750/10	1000/10
Mandelbrot	7500×10000	10000×10000	12500×12500

Experiment 1

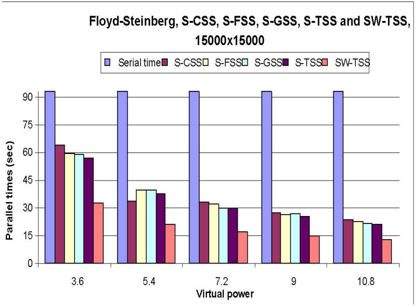
Speedups of the synchronized–only algorithms for Floyd-Steinberg

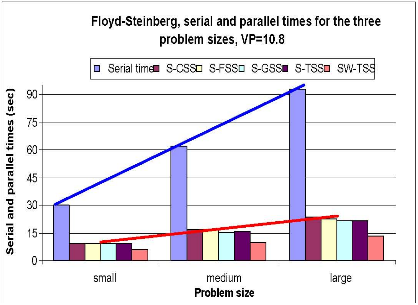
Test case	VP	$\mathscr{S}$-CSS	$\mathscr{S}$-FSS	$\mathscr{S}$-GSS	$\mathscr{S}$-TSS	$\mathscr{SW}$-TSS
Floyd-Steinberg	**3.6**	1.45	1.57	1.59	1.63	2.86
	5.4	2.76	2.35	2.33	2.47	4.35
	7.2	2.81	2.92	3.09	3.10	5.39
	9	3.41	3.50	3.49	3.70	6.27
	10.8	3.95	4.07	4.27	4.34	7.09

- The serial time was measured on the fastest slave type, i.e., twin
- $\mathscr{S}$-CSS, $\mathscr{S}$-FSS, $\mathscr{S}$-GSS and $\mathscr{S}$-TSS give significant speedups
- $\mathscr{SW}$-TSS gives an even greater speedup over all synchronized–only algorithms ☺ expected!

Experiment 1

Parallel times of the synchronized–only algorithms for Floyd-Steinberg





Serial times increase faster than parallel times as the problem size increases ⇒ larger speedups for larger problems ☺ anticipated!

Experiment 2

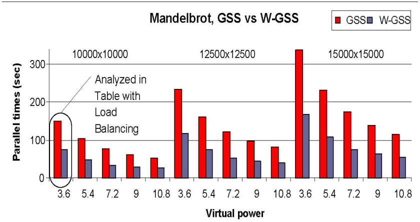
Gain of the weighted over non-weighted algorithms for Mandelbrot

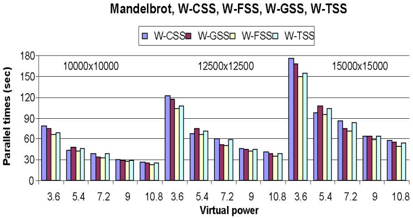
Test case	Problem size (large)	**VP**	CSS vs $\mathscr{W}$-CSS	GSS vs $\mathscr{W}$-GSS	FSS vs $\mathscr{W}$-FSS	TSS vs $\mathscr{W}$-TSS
Mandelbrot	15000×15000	**3.6**	27%	50%	18%	33%
		5.4	38%	54%	37%	34%
		7.2	45%	57%	53%	31%
		9	49%	54%	52%	35%
		10.8	46%	52%	54%	33%
Confidence interval (95%)	Overall 42 ± 3 %		40 ± 6 %	53 ± 6 %	42 ± 8 %	33 ± 4 %

- Gain is computed as $\frac{T_{\mathscr{A}} - T_{\mathscr{W}-\mathscr{A}}}{T_{\mathscr{A}}}$
- GSS has the best overall performance gain

Experiment 2

Parallel times of the weighted algorithms for Mandelbrot





The performance difference of the weighted algorithms is *much smaller* than that of their non-weighted versions ☺ anticipated!

Experiment 2

Load balancing obtained with $\mathscr{W}$ for Mandelbrot

Table: Load balancing in terms of total number of iterations per slave and computation times per slave, GSS vs $\mathscr{W}$-GSS.

Slave	GSS # Iterations (10^6)	GSS Comp. time (sec)	$\mathscr{W}$-GSS # Iterations (10^6)	$\mathscr{W}$-GSS Comp. time (sec)
twin2	56.434	34.63	55.494	62.54
kid1	18.738	138.40	15.528	62.12
twin3	10.528	39.37	15.178	74.63
kid2	14.048	150.23	13.448	61.92

$\mathscr{W}$-GSS achieves better load balancing and smaller parallel time

Experiment 3

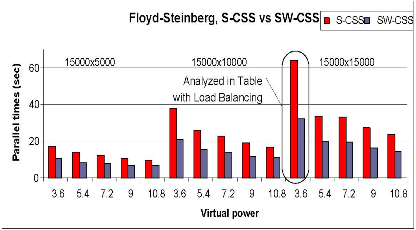
Gain of the synchronized–weighted over synchronized–only algorithms for Floyd-Steinberg

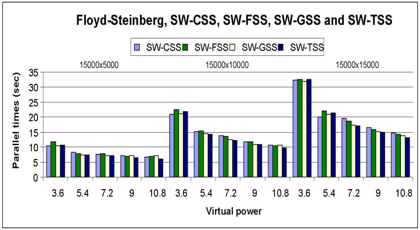
Test case	Problem size	VP	$\mathscr{S}$-CSS vs $\mathscr{SW}$-CSS	$\mathscr{S}$-GSS vs $\mathscr{SW}$-GSS	$\mathscr{S}$-FSS vs $\mathscr{SW}$-FSS	$\mathscr{S}$-TSS vs $\mathscr{SW}$-TSS
Floyd-Steinberg	15000×10000	**3.6**	**50%**	46%	45%	43%
		5.4	41%	48%	44%	43%
		7.2	41%	42%	41%	42%
		9	39%	43%	40%	41%
		10.8	38%	36%	38%	39%
Confidence interval (95%)	Overall 40 ± 1 %		39 ± 2 %	40 ± 3 %	40 ± 2 %	41 ± 2 %

- Gain is computed as $\frac{T_{\mathscr{S}-\mathscr{A}} - T_{\mathscr{SW}-\mathscr{A}}}{T_{\mathscr{S}-\mathscr{A}}}$
- CSS has the highest performance gain 50%

Experiment 3

Parallel times of the synchronized–weighted and synchronized–only algorithms for Floyd-Steinberg





The performance difference of the synchronized–weighted algorithms is *much smaller* than that of their synchronized–only versions

☺ anticipated!

Experiment 3

Load balancing obtained with $\mathscr{SW}$ for Floyd-Steinberg

Table: Load balancing in terms of total number of iterations per slave and computation times per slave, $\mathscr{S}$-CSS vs $\mathscr{SW}$-CSS

Test	Slave	# Iterations (10^6)	Comp. time (sec)	# Iterations (10^6)	Comp. time (sec)
		$\mathscr{S}$-CSS	$\mathscr{S}$-CSS	$\mathscr{SW}$-CSS	$\mathscr{SW}$-CSS
Floyd-Steinberg	twin2	59.93	19.25	89.90	28.88
	kid1	59.93	62.22	29.92	30.86
	twin3	59.93	19.24	74.92	24.06
	kid2	44.95	46.30	29.92	29.08

$\mathscr{SW}$-CSS achieves better load balancing and smaller parallel time than its synchronized–only counterpart ☺ anticipated!

Outline

Conclusions

- DOACROSS loops can be dynamically scheduled using $\mathscr{S}$
- Self-scheduling algorithms are quite efficient on heterogeneous dedicated & non-dedicated systems using $\mathscr{W}$
- $\mathscr{S}\mathscr{W}$ Self-scheduling algorithms are even more efficient on heterogeneous dedicated & non-dedicated systems

Future Work

1. Design a fault tolerant mechanism for the scheduling DOACROSS loops to increase system reliability and maximize resource utilization in distributed systems
2. Employ the scheduling algorithms presented earlier to perform large scale computation (containing both DOALL and DOACROSS loops) on computational grids
3. Use the scheduling algorithms presented earlier to schedule and load balance divisible loads (i.e. loads that can be modularly divided into precedence constrained loads)

Thank you for your attention!

Questions?

Mandelbrot

```
for (hy=1; hy<=hyres; hy++)  { /* scheduling dimension */
    for (hx=1; hx<=hxres; hx++)  {
        cx = (((float)hx)/((float)hxres)-0.5)/magnify*3.0-0.7;
        cy = (((float)hy)/((float)hyres)-0.5)/magnify*3.0;
        x = 0.0; y = 0.0;
        for (iteration=1; iteration<itermax; iteration++)  {
            xx = x*x-y*y+cx;
            y = 2.0*x*y+cy;
            x = xx;
            if (x*x+y*y>100.0)  iteration = 999999;
        }
        if (iteration<99999)  color(0,255,255);
        else color(180,0,0);
    }
}
```

Floyd-Steinberg Error Dithering

```
for (i=1; i<width; i++){ /* synchronization dimension */
    for (j=1; j<height; j++){ /* scheduling dimension */
            I[i][j] = trunc(J[i][j]) + 0.5;
            err = J[i][j] - I[i][j]*255;
            J[i-1][j] += err*(7/16);
            J[i-1][j-1] += err*(3/16);
            J[i][j-1] += err*(5/16);
            J[i-1][j+1] += err*(1/16);
    }
}
```

Modified LL23 - Hydrodynamics kernel

```
for (l=1; l<=loop; l++) { /* synchronization dimension */
    for (j=1; j<5; j++)  {
        for (k=1; k<n; k++){ /* chunk dimension */
            qa = za[l-1][j+1][k]*zr[j][k] + za[l][j-1][k]*zb[j][k] +
                 za[l-1][j][k+1]*zu[j][k] + za[l][j][k-1]*zv[j][k] +
                 zz[j][k];
            za[l][j][k] += 0.175 * (qa - za[l][j][k] );
        }
    }
}
```

Bibliography

Lamport, L.
The Parallel Execution of DO Loops.
Comm. of the ACM, 37(2):83–93, 1974.

Moldovan, D. I. and Fortes, J.
Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays.
IEEE Transactions on Computers, C-35(1):1–11, 1986.

Shang, W. and Fortes, J.A.B.
Time Optimal Linear Schedules for Algorithms with Uniform Dependencies.
IEEE Transactions on Computers, 40(6):723–742, 1991.

Darte, A. and Khachiyan, L. and Robert, Y.
Linear Scheduling is Nearly Optimal.
Par. Proc. Letters, 1(2):73–81, 1991.

Bibliography (cont.)

Koziris, N. and Papakonstantinou, G. and Tsanakas, P.
Optimal time and efficient space free scheduling for nested loops.
The Computer Journal, 39(5):439–448, 1996.

Varvarigou, T. and Roychowdhury, V. P. and Kailath, T. and Lawler, E.
Scheduling In and Out Forrests in the Presence of Communication Delays.
IEEE Trans. of Par. and Dist. Comp., 7(10):1065–1074, 1996.

Jung, H. and Kirousis, L. and Spirakis, P.
Lower Bounds and Efficient Algorithms for Multiprocessor Sceduling of DAGs with Communication Delays.
1st ACM SPAA, 1989.

Bibliography (cont.)

Chretienne, P.
Task Scheduling with Interprocessor Communication Delays.
European Journal of Operational Research, 57:348–354, 1992.

Andronikos, T. and Koziris, N. and Tsiatsoulis, Z. and Papakonstantinou, G. and Tsanakas, P.
Lower Time and Processor Bounds for Efficient Mapping of Uniform Dependence Algorithms into Systolic Arrays.
Journal of Parallel Algorithms and Applications, 10(3-4):177–194, 1997.

Papakonstantinou, G. and Andronikos, T. and Drositis, I.
On the Parallelization of UET/UET-UCT Loops.
Neural, Parallel & Scientific Computations, 2001.

Bibliography (cont.)

King, C.-T., Chou, W.-H., and Ni, L.
Pipelined Data-Parallel Algorithms: Part II Design.
IEEE Trans. on Parallel and Distributed Systems, 2(4):430–439, 1991.

J.-P. Sheu and T.-S. Chen
Partitioning and Mapping Nested Loops for Linear Array Multicomputers.
Journal of Supercomputing, 9:183–202, 1995.

Tsanakas, P. and Koziris, N. and Papakonstantinou, G.
Chain Grouping: A Method for Partitioning Loops onto Mesh-Connected Processor Arrays.
IEEE Transactions on Parallel and Distributed Systems, 11(9):941–955, 2000.

Bibliography (cont.)

- Drositis, I. and Goumas, G. and Koziris, N. and Tsanakas, P. and Papakonstantinou
  Evaluation of Loop Grouping Methods based on Orthogonal Projection Spaces.
  Proc. Int. Conf. on Par. Proc. ICPP-2000, 469–476, 2000.
- Papadimitriou, C. and Yannakakis, M.
  Toward an Architecture-Independent Analysis of Parallel Algorithms.
  SIAM J. of Comp., Ext. Abstract in Proc. STOC 1988, 19:322–328, 1988.
- Wolf, M. E. and Lam, M. S.
  A data locality optimizing algorithm.
  PLDI '91: Proc. of the ACM SIGPLAN 1991 Conf. on Progr. Lang. Design and Impl., 30-44, 1991.

Bibliography (cont.)

J. Ramanujam and P. Sadayappan
Tiling Multidimensional Iteration Spaces for Multicomputers.
Journal of Parallel and Distributed Computing, 16(2):108–230, 1992.

Boulet, P. and Darte, A. and Risset, T. and Robert, Y.
(Pen)-ultimate tiling.
Integration, the VLSI Journal, 17:33–51, 1994.

Jingling Xue
On Tiling as a Loop Transformation.
Parallel Processing Letters, 7(4):409-424, 1997.

Goumas, G. and Drosinos, N. and Athanasaki, M. and Koziris, N.
Message-Passing Code Generation for Non-rectangular Tiling Transformations
Parallel Computing, 32(11), 2006.

Bibliography (cont.)

- Andronikos, T. and Kalathas, M. and Ciorba, F. M. and Theodoropoulos, P. and Papakonstantinou, G.
  An Efficient Scheduling of Uniform Dependence Loops.
  6th Hellenic European Research on Computer Mathematics and its Applications (HERCMA'03), 2003.

- Andronikos, T. and Kalathas, M. and Ciorba, F.M. and Theodoropoulos, P. and Papakonstantinou, G.
  Scheduling nested loops with the least number of processors.
  Proc. of the 21st IASTED Int'l Conference APPLIED INFORMATICS, 2003.

- Ciorba, F. M. and Andronikos, T. and Papakonstantinou, G.
  Adaptive Cyclic Scheduling of Nested Loops.
  7th Hellenic European Research on Computer Mathematics and its Applications (HERCMA'05), 2005.

Bibliography (cont.)

Ciorba, F. M. and Andronikos, T. and Drositis, I. and Papakonstantinou, G. and Tsanakas, P.
Reducing the Communication Cost via Chain Pattern Scheduling.
4th IEEE Conference on Network Computing and Applications (NCA'05), 2005.

Ciorba, F. M. and Andronikos, T. and Kamenopoulos, D. and Theodoropoulos, P. and Papakonstantinou, G.
Simple code generation for special UDLs.
1st Balkan Conference in Informatics (BCI'03), 2003.

Bibliography (cont.)

- Andronikos, T. and Ciorba, F. M. and Theodoropoulos, P. and Kamenopoulos D. and Papakonstantinou, G.
  Code Generation For General Loops Using Methods From Computational Geometry.
  IASTED Parallel and Distributed Computing and Systems Conference (PDCS'04), 2004.
- Ciorba, F. M. and Andronikos, T. and Riakiotakis, I. and Chronopoulos, A. T. and Papakonstantinou, G.
  Dynamic Multi Phase Scheduling for Heterogeneous Clusters.
  20th International Parallel and Distributed Processing Symposium (IPDPS'06), 2006.

Bibliography (cont.)

Papakonstantinou, G. and Riakiotakis, I. and Andronikos, T. and Ciorba, F. M. and Chronopoulos, A. T.
Dynamic Scheduling for Dependence Loops on Heterogeneous Clusters.
Neural, Parallel & Scientific Computations, 14(4):359–384, 2006.

Ciorba, F. M. and Riakiotakis, I. and Andronikos, T. and Papakonstantinou, G. and Chronopoulos, A. T.
Enhancing self-scheduling algorithms via synchronization and weighting.
J. Parallel Distrib. Comput., 68(2):246–264, 2008.

Bibliography (cont.)

Ciorba, F. M. and Riakiotakis, I. and Andronikos, T. and Chronopoulos, A. T. and Papakonstantinou, G.
Studying the Impact of Synchronization Frequency on Scheduling Tasks with Dependencies in Heterogeneous Sytems.
16th International Conference on Parallel Architectures and Compilations Techniques (PACT '07), 2007.

C.P. Kruskal and A. Weiss
Allocating independent subtasks on parallel processors.
IEEE Trans. on Software Eng., 11(10):1001–1016, 1985.

T.H. Tzen and L.M. Ni.
Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers.
IEEE Trans. on Par. and Dist. Sys., 4(1):87–98, 1993.

Bibliography (cont.)

A.T. Chronopoulos, R. Andonie, M. Benche and D. Grosu.
A Class of Distributed Self-Scheduling Schemes for Heterogeneous Clusters.
3rd IEEE Int. Conf. on Cluster Computing, 2001.

S.F. Hummel, E. Schonberg and L.E. Flynn.
Factoring: A Method for Scheduling Parallel Loops
Comm. of the ACM, 35(8):90–101, 1992.

C.D. Polychronopoulos and D.J. Kuck.
Guided self-scheduling: A practical self-scheduling scheme for parallel supercomputers
IEEE Trans. on Computer, C-36(12): 1425–1439, 1987.